2003-06

Agent-based simulation of robotic systems

Williams, Manoleto Z.

Monterey, California. Naval Postgraduate School

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

AGENT-BASED SIMULATION OF ROBOTIC SYSTEMS

by

Manoleto Z. Williams

June 2003

Thesis Advisor:	Richard Harkins
Second Reader:	John Hiles

This thesis done in cooperation with the MOVES Institute

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2003	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: Agent-Based Simulation of Robotic Systems			**5. FUNDING NUMBERS**
6. AUTHOR(S) Manoleto Z. Williams			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT ***(maximum 200 words)***

A goal and behavior agent layer Java Model was developed to simulate cruise, correct and avoid Control Modules in an autonomous agent (robot). The model was tested against a deterministic Figure of Merit (FOM) to predict a "best mix" of agents for the simplistic agent economy parameters given. Future works suggests validation of the model with real agents in a real economy.

14. SUBJECT TERMS Agent Based Simulation, Modeling and Simulation, Robotics, Self-organization, Behavior Based Robotics			**15. NUMBER OF PAGES** 181
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

AGENT-BASED BEHAVIORAL SIMULATION OF ROBOTIC SYSTEMS

Manoleto Z. Williams
Lieutenant, United States Navy
B.S., United States Naval Academy, 1996

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MODELING, VIRTUAL ENVIRONMENTS, AND SIMULATION (MOVES)

from the

NAVAL POSTGRADUATE SCHOOL
June 2003

Author: Manoleto Z. Williams

Approved by: Richard Harkins
Advisor

John Hiles
Second Reader

Rudy Darken, Chair, Academic Committee
Modeling, Virtual Environments, and Simulation Group

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

A goal and behavior agent layer Java Model was developed to simulate cruise, correct and avoid Control Modules in an autonomous agent (robot). The model was tested against a deterministic Figure of Merit (FOM) to predict a “best mix” of agents for the simplistic agent economy parameters given. Future works suggests validation of the model with real agents in a real economy.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

This thesis is dedicated to Space Systems Center San Diego who provided funding. Additional thanks go to:

- Professor Richard Harkins for his role in immersing me in the robotics community.
- Professor John Hiles for his continued support and guidance in Multi-Agent Systems.
- Bart Everett for his support and guidance in the robotics and simulation community.
- Katherine Mullens for her continued support as a liaison with Space Systems Center San Diego.

Lastly, and most importantly, a very special thanks to my lovely wife Heather for her undying love and support and my wonderful children Zamon, Tegan, and Exia.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. THESIS STATEMENT

Agent-Based control of mobile robots can be accomplished using the techniques of resource management, and provide a scalable architecture for autonomous robotic entities.

B. MOTIVATION

Intelligent agents have largely resided in software systems. Their use in simulation has provided a level of fidelity that enables emulation of human reasoning and action. Often agents are used to perform some task that would normally require a human operator to accomplish. Using agent-based simulation to explore new configurations in robotic systems will allow the engineer of an actual platform to reason about runtime configurations prior to a complete system build.

Multi Agent Systems offer a tool for exploring emergent behavior in software systems that can be used to make inferences and hypothesis about the real world. Autonomous agents are able to execute their own plans based on their beliefs about the environment, their desires to accomplish certain task, and their intentions as to how to accomplish the tasks at hand. Observing the behaviors simulated from environmental cues, their reactionary behaviors can lead to discovery of unexpected results that may be costly in a real robotic system.

Robotic systems have traditionally been implemented with a set of engineered algorithms and sensing techniques that allow the robot to perform a limited set of behaviors. Coupling a number of behaviors in one robotic system or amongst a number of robotic systems can produce an emergent behavior emulating low-level species tasks such as foraging or swarming. By continually adding behaviors to these systems or system of systems, we can move towards evolving the systems into complex adaptive systems. The first step in formulating what behaviors to invoke in a system can be implemented in simulation. With a robust set of behaviors tested in simulation, the overhead of reworks in a robotic system can be reduce and in turn save time and money.

Fusing the techniques of agent-based intelligence with the physical attributes of a robotic system allows the robotics researcher to explore new ways to employ intelligent robots. Robots can be equipped with the same architecture embodied in an intelligent agent to have its own beliefs, desires, and intentions while performing a real world task.

In order to bridge the gap between simulation and real robotic systems, the simulation must meet a criterion that is experienced in the real world. This is not to say that the simulation must completely model the real world, however, many situations may be taken for granted in the simulation that cannot be afforded in the real world. This thesis attempts to define an agent-based system that employs the use of resource management with an economy that takes into account many limitations in real robotic systems. Robots are limited by power, size, payload and time on station. Many of these factors make it physically impossible for resource limited robotic platforms to employ techniques such as swarming over large areas of terrain. My motivation for this thesis is to research agent-based techniques for allocating resources to address the movement of robotic platforms in a coordinated fashion to accomplish a common goal.

C. ROBOT USES

Futurist and Hollywood film-makers have found numerous uses for robots from domestic housework to surgery. Some uses have been realized and brought to fruition in commercial robots such as the Land Shark, which conducts simple collision avoidance while mowing a lawn on its path (an RF fence keeps it within the boundaries of the lawn). Other robots have found themselves deployed in military operations controlled by human operator via a radio frequency (RF) base unit to provide visual feedback to the operator. In each case, the robot provides some assistance to the user. Robot uses typically fall under one or more of the following three D's, jobs that are dirty, dull, or dangerous.

Robots can be used to work in areas that are polluted or contaminated and would require humans to conduct the task wearing bulky protection suits that in some cases limit the persons ability to perform the task. Robots have been used in the nuclear industry for environmental restoration of irradiated and polluted sites. In space robots can also be used in foraging activities and to explore areas of uncertainty.

Dull tasks include the tasks in industry, which are mundane for human workers, such as assembly lines. Certainly the auto industry has capitalized on the idea using robotic actuators to construct vehicles for quite some time now. For many repetitive processes that require the same precise movement, robots have proven to be more useful than their human counterparts due to lack of concentration and fatigue in the human operator. These menial tasks lend themselves well to a robotic system.

With the changing face of today's battlefield, robots can provide a tremendous capability in an effort to preserve human life. Searching for, detecting, and clearing unexploded ordinance is an excellent use of robotic systems and provides an opportunity to save the lives of men and women who would otherwise be tasked with such a dangerous mission. Other applications in a dangerous scenario are search and rescue of survivors in an unstable disaster area, such as terrorist attacked building or searching for booby traps.

There exists a wide range of applications in which robotics can greatly increase the productivity or preservation of life in today's world. While many have envisioned the ubiquitous use of robotics, it is still a large undertaking for the robotics researcher and engineer to bring these very capable systems to reality. Much of the work ahead relies on the ability to find the right mix of robotic systems that we can incorporate into our daily lives.

D. NEED FOR AUTONOMOUS ROBOTS

Autonomous robots provide a wealth of resources by handling tasks that are normally completed by their human counterparts. The use of a robotic system is envisioned to relieve the human operator of doing tasks that are menial or inherently dangerous. If a menial task normally conducted by human can now be completed by a human controlling a robot, not much ground has been gained since the human operator will find himself simply conducting a menial task using a robot. The true benefit of a robotic system is its employment and ultimate reduction of human intervention with the job that the robot has been assigned to complete.

Current military applications of unmanned systems involve the tele-operation of robotic systems to conduct some of the more dangerous tasks normally conducted by

military personnel. This capability has provided a tremendous enhancement to units that are forward deployed in areas such as Afghanistan. Autonomy ensures that a large number of robotic systems can be deployed while reducing the number of human operators required operating the systems. Autonomous robots are also inexpensive and disposable compared to the cost of deploying and in some cases losing personnel in the battlefield.

E. ECOLOGICAL NICHE

A plant's or animal's niche, or more correctly, ecological niche, is a way of life that is unique to that species. The idea of an ecological niche also holds true in robotics. Certainly a robot designed to deliver mail in an office building would be of little use on the battlefield searching for land mines. Designing robots requires careful consideration of the ecological niche of the robot. What environments will the robot be used in, and how will the robot function in that environment? Robots are situated agents operating in an ecological niche. They are an integral part of the world, and when they act, they affect the world and can receive immediate feedback about the world in which they have acted upon. In developing a robotic system careful consideration must be given as to what actions a robot will take in an environment and what affect should most likely result from that action.

Biological species exhibit this behavior based on their own ecological niche. For example, the red fox's habitat might include forest edges, meadows and the bank of a river. The niche of the red fox is that of a predator, which feeds on the small mammals, amphibians, insects, and fruit found in this habitat. Red foxes are active at night. They provide blood for black flies and mosquitoes, and are host to numerous diseases. The scraps, or carrion, left behind after a fox's meal, provide food for many small scavengers and decomposers. This then is the ecological niche of the red fox. Only the red fox occupies this niche in the meadow-forest edge communities. In other plant communities, different species of animal may occupy a similar niche to that of the red fox. For example, in the grassland communities of western Canada and the United States, the coyote occupies a similar niche (to that of the red fox.).

As for a robotic system, a robot may be tasked with sorting and delivering mail to different offices. As it journeys from one office to the next, it avoids collision with other

objects both static and dynamic. This may be done with some sort of vision sensor or range sensor such as active sonar. The robot has some type of ontology to discern floors, elevators, office numbers or room numbers assigned to personnel. This then is the ecological niche of the mail delivery robot. The obstacle avoidance algorithm the robot uses works well for its environment of an office building. The ontology used carries its merit in that of an office setting. Physical design of the robot works well in the office setting such as the wheel types, the payload and power considerations.

Robots are typically designed from an ecological niche standpoint by first answering the question, "What do I want this robot to be able to do?" Once this question is answered the roboticist is able to draw from a number of techniques to develop a system unique to its niche target.

F. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

Chapter II, Background: Research in various areas similar to the Agent Economy either through natural systems, agent-related work, AI, or military simulation. Description of existing and past research, which areas were complementary, and which were inapplicable.

Chapter III, Architecture: Describes the base design of the single robotic unit used in research for this thesis, and discusses the scaling methods for using multiple robots cooperates to accomplish a common goal. Further time is spent on discussing some of the basic behaviors of the system that are used in the Agent Economy simulation

Chapter IV, Implementation: Describes the basic structure of the Java program running the Agent Economy simulation and the data structures on which it is built. Then details the modules created for this thesis, how they interact with the underlying code, and how they implement the decisions made in Chapter III.

Chapter V, Analysis and Results: Experimental design for testing of the system and analysis of how well it met its design goals. Conduct of the actual experiment, and what results were derived.

Chapter VI, Conclusions and Future Work: Discussion of the strong and weak points of the system as currently implemented, and suggested directions for future research and development.

II. BACKGROUND

A. INTRODUCTION

In this Chapter, I give an overview of topics related to this thesis. Some topics discussed here will be mentioned in more detail than others, since they are more closely related to my research. I organize this chapter as collection of short introductions. Instead of simply stating the literature, I will comment on some aspects of the emphasized topic. Furthermore, I will also investigate possible future applications and state the differences from the approach taken in this work, whenever necessary. It is my intention to keep this chapter as interesting as possible.

B. SELF-ORGANIZATION

1. Definition

The term "self-organization" (or "self-organizing system," to be precise) is first defined by Farley and Clark of Lincoln Laboratory in 1954:

> A self-organizing system is a system that changes its basic structure as a function of its experience and environment.

This definition clearly relates to today's "hot" topics of adaptive control, neural networks and genetic algorithms. I will also dwell upon neural networks and unsupervised learning briefly at the end of this chapter. In this thesis self-organization refers to the way in which agents can adapt to changes in their environment and adjust their behaviors to accomplish a goal or a task.

A self-organizing system has three main characteristics (or functions) (Selfridge, 1962):

- Affect
- Telos
- Effect

To explain these three functions, I will use my Agent-Economy scenario as an example: A robotic agent in a multi-robot network observes its environment (affect), uses these observations to decide what to do next (telos) and then executes according to this decision (effect).

In a population of Agent-Economy robots dispersed in an area where several objects are located, agents recognize the situation by observing the signals coming from other agents and "goals" (affect), compute the direction of movement at each step (telos) and then move (effect) based on this information. On a large scale, the whole population receives signals (affect) and then, guided by decision algorithms (telos), acts (effect) appropriately.

One encounters self-organization in many fields:

- ecology (insect societies, ecosystems)
- Chemistry (thermodynamics)
- Computer science (decision algorithms, neural networks and fuzzy logic)
- Geology (tectonic movements)
- Sociology (communication and migration)
- Economy (socio-spatial systems

Nicolis and Prigogine, defining self-organization in non-equilibrium systems, stated that self-organization emphasizes the large scale coordination processes at many levels (Nicolis and Progogine, 1977). Nonlinear processes and non-equilibrium conditions play a significant role in these processes. Kauffman believes that self-organization, an "inherent property of some complex systems," may be responsible for biological evolution along with selection (Kauffman, 1991). His computer models suggest that certain complex biological systems tend toward self-organization.

2. Characteristics of Self-Organizing Systems

Self-organization has three important characteristics:

- First, a self-organizing system can accomplish complex tasks with simplistic individual behavior.
- Secondly, a change in the environment may influence the same system to generate a different task, without any change in the behavioral characteristics.
- Finally, any small differences in individual behavior can influence the collective behavior of the system.

Therefore, social complexity of the system is compatible with simple and identical individuals, as long as communication among the members can provide the necessary amplifying mechanism. For example, as I mention in Chapter 4IV, our

"swarm" of robotic agents gathering to interrogate an object, can change their operational goals by a signal from any member of the group. This can be achieved by defining specific communication mechanisms.

In a self-organizing system, individual behavior need not be changed in order to have different collective behavior. This characteristic of self-organization is highly advantageous for a swarm of robots since simple individual behavior can be achieved with relatively cheap and simple designs.

3. Advantages of Self-Organization

What makes a self-organizing system advantageous over a preprogrammed, deterministic organization is that the former is based on individuals/agents requiring simple programming and autocatalytic communications. A large number of individuals can be coordinated into a collective system interacting with their environment. And as stated above, this collective behavior will have an "adaptive" characteristic. Such a system is therefore simple, reliable and adaptive while only a few basic rules are needed to define individual behavior and interactions.

Some animal societies and particularly social insects can achieve complex tasks that are impossible to complete individually. I will state some examples in the next section. On the other hand, simplicity (and homogeneity) of individual agents in a robotic swarm decreases the cost of production and the likelihood of the breakdown.

Furthermore, breakdown of one agent will not affect the activity of the whole robotic team, which may not be the case in a deterministic system such as a production chain. The simplicity would also be in software as well as in hardware. In a deterministic system, programs are highly complex, in order to operate in every possible situation harmful to the system, and it is still impossible to foresee them all. However in a self-organizing system, simpler programs can operate in unforeseen situations and adapt to changing conditions. For these reasons, self-organizing algorithms, which have only partial (local) knowledge of the network, are used to manage data networks of large numbers of users.

Advantages of self-organization and the efficiency in self-organizing behavior of some animal societies, as they became known, caused interest in the use of self-organization in robotics. To quote Deneubourg and Goss:

> Engineers are often, consciously or not, prisoners of the Cartesian and scientific positivist philosophy that dominates their education, and it is therefore not surprising that robot designers have chosen to develop expensive, complicated, deterministic robots, tailored to specific problems. We can now propose the completely different approach of using teams of simple, interacting robots to perform a wide range of tasks.

As engineering society becomes more interested in adaptive, decision-making systems such as neural nets, fuzzy logic, etc., it is obvious that this approach will draw more attention in the future.

C. NATURAL SYSTEMS

Some animal societies such as colonies of ants and bees, flocks of birds, schools of fish, can be an inspiring model for a self-organizing robotic network. In this section, I will summarize some interesting characteristics of above-mentioned animal societies.

1. Schools of Fish

Another interesting self-organized behavior is found in schools of fish. Hundreds of fish, moving like a single organism, can disperse in a quick expansion in case of a danger (in form of a bigger fish probably) and then group again to reform the school. Schooling serves to reduce the risk of being eaten for a fish, since the probability of detection is reduced by forming a school. Also even if a school is detected by a predator, the odds of being eaten is still less for an individual fish (Partridge, 1982).

Although most work done on schools of fish studied species of fish that are consumed, some predators also form schools. If a member of the school finds food, the other members can take advantage of the find. If the members of the school remain barely in the sight of one another, the search area is at a maximum. Application of this idea to populations of multiple mobile robots for searching pollutants, for planetary missions or for detecting missile launches, is obvious.

Partridge determined an interesting coordination in tuna schools. Tuna schools of 50 or more members sometimes divide into smaller groups which consist of between 10

and 20 fish. These fish spread out along a curve very similar to a parabola with concave side forward. Although achieving a regular distance between individuals along a parabola is difficult, that form provides a considerable advantage in hunting. If the parabolic school swims parallel to its axis, any prey reacting to the curved school, will be driven to the focus of the parabola, which is the most convenient place for surrounding the prey.

Fish schools do not have a regular geometric form; the structure is loose or probabilistic and it results from each fish's applying a few simple behavior rules. First rule is that each individual maintains an empty space around itself. In general, only one neighbor at a time is at the preferred distance from a particular fish. (In a regular geometric shape, neighboring fishes would be at the same distance.) Fish also tend to keep their neighborhood at a particular preferred angle with respect to their body angle. Most schools of fish are organized on the same lines: preferred distance and angle.

Experiments on pollock (Partridge, 1982) showed that vision and lateral lines are two important senses fishes employ to match the speed and direction of other fish. Blinded fish and fish whose lateral lines are removed were able to school. But blinded fish swam farther from their nearest neighborhoods than pollock's ordinarily do, while fish with lateral lines removed swim closer to the nearest schoolmate. Only when a fish was both blinded and had had its lateral lines removed it did fail to maintain its position in the school. Vision seemed to provide the “attractive force” between members while lateral lines provided the “repulsive force.” Other research suggested that vision takes precedence in case of contradictory information.

2. Flocks of Birds

Flocks of birds are organized more or less the same way as the schools of fish. Each member of the flock is attracted to the flock; at the same time, they are repelled from other member in the vicinity by an obstacle avoidance “goal.” Computer simulations based on three simple rules, could create flocks of birds which seemed to correspond to our notion of what constitutes flock-like motion (Reynolds, 1987). In order of precedence, these are:

1. Collision avoidance
2. Velocity matching with nearby flock-mates
3. Flock centering in attempt to stay close to nearby flock-mates

3. Termites

Another highly interesting self-organization example is encountered in termites: the periodic assembling of a nest by a population (Kugler and Turvey, 1987). The nest building behavior of termites consists of several distinct phases of construction. In the first phase, building material are carried into the site and deposited randomly. This phase ends when preferred sites, which are fewer than original deposits, emerge. In the next phase, material buildup continues until deposit sites take the shape of pillars. When pillars reach certain size, third phase of construction starts. Two neighboring pillars mutually bend toward a virtual midpoint. End of the third phase is defined by formation of an arch. And in the final phase, construction of an arching dome that extends from the tops of arches takes place. These phases can be repeated on top of the dome if random deposition of material begins again.

The formation of this complex structure involves pheromones. The insects follow two simple rules:

1. Move in the direction of strongest smell
2. Deposit where the smell is strongest

Each deposit creates an “aromatic potential field.” Because the number of insects is large, the likelihood that an insect will move in the direction of a recent deposit will increase. The more attractive a site becomes because of increasing pheromone concentration, the more frequent the deposits (of material and, therefore, pheromones) on that site, which in turn increases the pheromone concentration. This sequence requires a certain number of insects. Only above a critical number of insects, can the pheromone amplify and become effective, since it has a diffusive character.

When a pillar develops on a site of an original deposit, its uppermost region, being the deposition point, acts as a point attractor for insects. When two pillars are

sufficiently close to each other, a virtual saddle point midway between the pillars results. Therefore, insects first approach the saddle point and then converge to one of the pillars from the direction of the other. This behavior leads to the formation of an arch. And the formation of arches, creating new attraction points, can result new saddle points that guarantee the formation of a dome. The cycle can repeat when new deposit sites emerge on top of the dome.

D. MULTIPLE MOBILE ROBOTS

In this section, some previous work on autonomous mobile robots, multi-robot systems and robot behavior will be cited. I will try to highlight important ideas and significant achievements on the above-mentioned fields.

1. Autonomous Mobile Robots

Since autonomous mobile robots are the basic elements of multiple mobile robot populations, I will first dwell upon autonomous mobile robots. Subsumption control architecture and several navigation techniques will be summarized in this subsection.

a. Subsumption Architecture

Subsumption architecture for controlling mobile robots was first introduced by Brooks (Brooks, 1986). In such architecture, layers of control system are built in order to let the robot operate at increasing levels of competence. Layers are made up of asynchronous modules that communicate over low-bandwidth channels. Each module is a simple computational machine, and higher-level layers can suppress the output of lower levels (subsumption). But, lower levels continue to function as higher levels, which interfere with their data inputs, are added. Check alignment all the way through

Each level generates a behavior and the competence of the robot is improved by addition of new layers. The subsumption architecture is based on decomposition of a mobile robot in terms of behavior rather than in terms of functional modules. Since the overall control system can be viewed as a system of agents acting separately, there is no need for a central control module.

An example of subsumption architecture is Squirt, a very small intelligent mobile robot. Squirt acts as a bug, hiding in dark corners and venturing out in the

direction of noises, only after noises are gone, looking for a new place to hide near where the previous set of noises came from. The most interesting fact about Squirt is the way in which its high-level behavior, mentioned above, emerges from a set of simple interactions with the environment. Squirt's lowest level of behavior causes the robot to search for darkness. The second level of behavior is triggered once a dark spot has been found. Monitoring two microphones, the direction from which the noises come is detected, and when a few minutes of silence follows a sharp pattern of noise, Squirt moves in the direction of the last heard noise, suppressing the desire to stay in the dark. After a time-period, the first level is no longer suppressed and becomes active. This "bug behavior" fits in 1300 bytes of code on an 8-bit microprocessor (Brooks, 1990).

The subsumption architecture has also demonstrated robust navigation for mobile robots in dynamically changing environments. Its layered structure is well-adaptable for hardware implementation.

b. Autonomous Navigation

The most important "function" (or the first layer of a subsumption control architecture) in a mobile robot is the ability to avoid obstacles, as it is in schools of fish and flocks of birds. An autonomous robot recognizes its environment using sensors and decides what to do next based on the sensor data. Rodin and Amin defined the general structure of an intelligent navigational algorithm for solving the problem of real time control in an environment with moving obstacles as follows: it consists of *identifier*, *goal selector* and *adapter* levels (Rodin and Amin, 1998).

- The *identifier* constructs a local representation of the surroundings based on information obtained from sensors, and determines the speed of obstacles.
- *Goal selector* uses the map and speed of the obstacles and finds a locally optimal collision-free path satisfying other possible conditions.
- The *adapter* consists of two subsystems: one for path smoothing to avoid sharp turns and the other for determination of steering command (based on potential field path planning).

Problems often encountered in autonomous navigation models are (i) delay in feedback information, (ii) sensor and servo errors, and (iii) limited sensor range (Feng and Krogh, 1989). Due to the large amount of computation required to process the

sensor data, a delay is expected in obtaining the local map. For Agent Economy robots, this would not be a problem since the simulation does not include any map and/or path finding algorithms. Again sensor and servo errors create a problem for map building robots. Limited sensor range may cause a problem in obstacle avoidance. However, it is possible to overcome this by adjusting the speed of rovers according to the visibility range of the sensors.

On the other hand, Arkin, describing path planning and navigation as a collection of behaviors, uses *motor schemas* to obtain a reactive navigation method for autonomous robots. Motor schemas serve as the basic unit of behavior specification for the navigation; they are concurrent processes that operate in conjunction with associated perceptual schemas and contribute independently to the overall action of the robot (Arkin, 1999). A variant of the potential field method is used to produce the appropriate velocity and steering commands. Motor schemas, such as *move-ahead*, *move-to-goal*, *avoid-obstacle*, which can be visualized as vector fields, are represented as asynchronous computing agents in terms of addition and multiplication. Figure 1 illustrates the logical inputs to the robots vector association field used to produce the final movement of the robot.

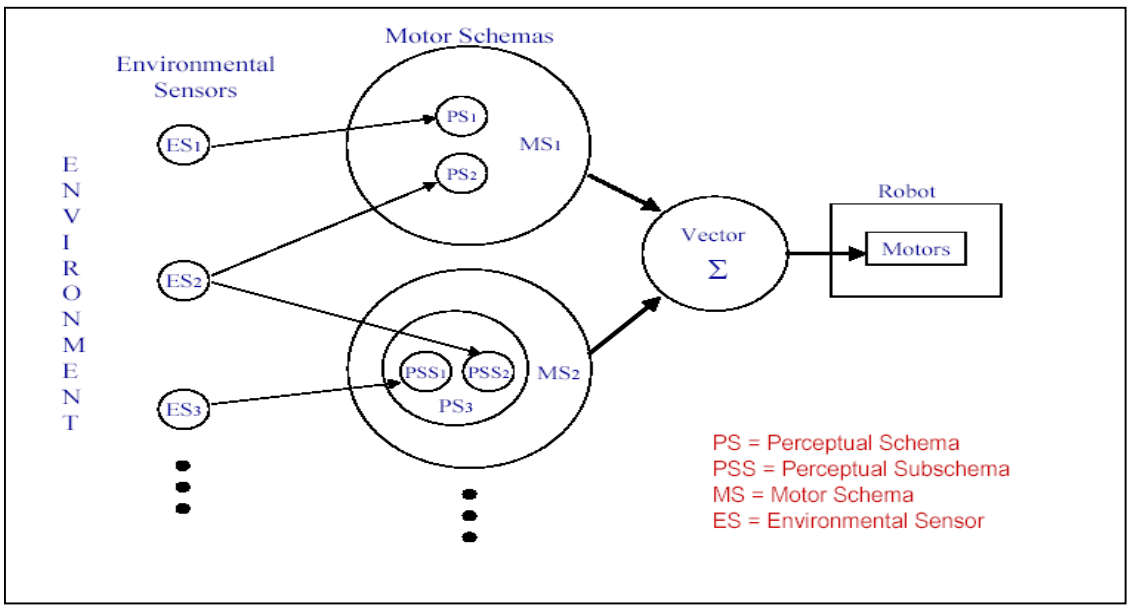

Figure 1. Motor Schema Diagram

The output of a schema is a single velocity vector derived from a potential field formulation of the forces exerted upon the robot at any particular point in space. The entire potential field is never computed; thus, the computational demand for a single schema is small. The output of each motor schema is combined using vector summation, and then normalized. Arkin's model includes a low-magnitude random vector that changes at random time intervals in order to remove the robot from undesirable equilibrium points that arise when active motor schemas balance each other. Also, gains of schema outputs can be changed (depending on established real-time deadlines for goal attainment) in order to allow a blocked robot to bypass obstacles. Arkin states that what might appear to be a naive approach, the summing of individual vector outputs of the "schemas," works quite well, both in simulations and real world experiments. Figure 2 shows two potential fields the first represents a repulsion field, the second represents the attraction field, and the third represents the resultant field once the two previous potential fields are added together giving the robot a path to follow based on the added vectors. The arrow indicates that the path of the robot is determined by a combination of repulsion from the first field and an attraction towards the second field.

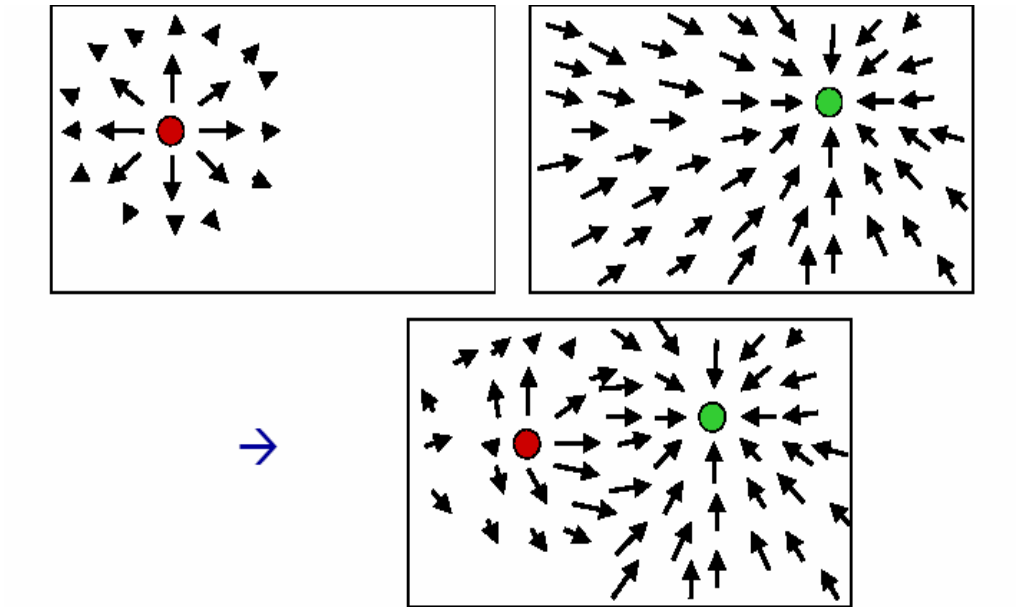

Figure 2. Vector Based Potential Field

Although most reactive systems are not concerned with the use of world knowledge (map), Arkin's autonomous robot architecture (Arkin, 1999) includes *a priori* information about the environment. The Agent-Economy scenario is closer to Brooks' works, which avoid "world modeling" for individual insect-like robots (such as Squirt and Genghis, a six-legged robot).

2. Multi-Robot Systems

In this section I will dwell upon some previous work done on coordination and control of multi-robot systems, excluding coordination of multiple manipulator systems - which is generally based on centralized control.

Systems of multiple mobile robots have gained interest in recent years when projects such as planetary surface mission and hazardous waste management, emerged. Large populations of mobile robots, as decentralized robotic systems (DRS), have many advantages over centralized systems, especially when high reliability is required, such as maintenance tasks in nuclear power plants.

The application of multiple mobile robots to planetary missions is outlined by Miller in (Miller, 1990). This work states the fact that teams of small autonomous robots have advantages, such as lower cost, lower launch/landing mass and mission reliability, over larger robots. Behavior driven control methods described in the previous section are likely to be used in designing such small robots. The use of fixed radio beacons is also anticipated along with the necessity of leader selection for formation of a coordinated team. Leader selection can be achieved by assigning serial number to robots. Robots are assumed to be able to transmit/detect these numbers, and the one with higher serial number will be collectively elected as leader.

Miller also emphasize the fact that coded beacons and beacon readers on each robot , with other simple broadcast signals, could be sufficient to achieve a complex task with individual behavior, since navigation and homing techniques are well developed for autonomous mobile robots.

The term "distributed robotic system" (DRS) is sometimes used to describe a multi-robot system based on instinctive responses and cooperation. Simulations of DRS designed for searching for pollutants are created by Genovese, et al., based on biological

systems and subsumption-like architectures(Genovese and others, 1992). This design suggests a supervising user who can localize agents whenever necessary. In this context, the system is not a "swarm." Communication between agents is more complex than the one described by Miller and includes coded radio transmission on a single channel. Although this model includes "acknowledge" messages, this type of communication proves to be advantageous, as we investigate in Chapter 4.

Beni and Wang claim that all agents involved in an operation must communicate to each other the intention to execute their part of operation (Beni and Wang, 1991). Stating that the "commitment protocols" are basic building blocks of distributed computing algorithms, such communication is defined as a required characteristic. On the contrary, biological systems described earlier in this chapter are able to operate as a self-organizing system without direct communication (of intentions). The key factor here is the large number of agents. But again, in the Agent-Economy scenario, there may be a situation where number of agents at each "goal" is not sufficient to start the next phase. A temporary "*do-not-consider-this-goal*" signal can be introduced by the *leader* to overcome this problem.

Previous experimental works on multiple mobile robots include ACTRESS (ACTor Based Robots and Equivalent Synthetic Systems) developed by Habib et al., and Yamabica robots realized by Yuta and Premvuti. ACTRESS, as an autonomous and decentralized robotic system, does not only have mobile robots, but also any kind of robotic system and/or computers. Mobile robots developed for ACTRESS have a portable computer instead of an onboard microprocessor and weigh 51 kg. They demonstrated intelligent navigation behavior.

Yamabica robots, more compact than above-mentioned robots, are used to determine a solution for a deadlock situation caused by multiple mobile robots with overlapping running courses in aisles. The method described provides a shunting process to solve the deadlock. However it requires constant broadcast of information (e.g., current position), a world knowledge, and is based on complex decision modules "managing" the information obtained from sensors. Both ACTRESS and Yamabico models differ from our approach to self-organization in many contexts.

Part of the work on self-organization in this thesis (Chapter III) is inspired by an interesting work by Sugihara and Suzuki. They give a method for motion coordination of a group of mobile robots. Each robot plans its motion individually based upon a defined goal and detected position of other robots. This method is fully distributed in that sense and shows that intelligent behavior can emerge from simple individual behavior.

Sugihara and Suzuki were able to create different geometric shapes by defining simple algorithms to be executed by a large number of agents. Their simulation shows that robots can form lines, circles, polygons and distributes themselves within a circle or convex polygon in the plane. Although the algorithms defined in are shown to work quite well, most of these algorithms are based on the assumption that each robot can detect the distance from the farthest teammate (as well as the distance from the nearest teammate). In this research, we use the advantage of a goal beacon and eliminate the need for the distance from the farthest agent in computations.

3. Military Robotic Systems

The robotics group at SPAWAR San Diego, the Complex Adaptive Systems Center, has been actively engaged in robotics research for the past twenty years. Many of their platforms originated as tele-operated platforms and have since born a number of autonomous robots able to navigate themselves to waypoints and avoid colliding with obstacles in their paths. In developing the platforms, many architectures were developed that support a robust set of communications between mobile robotic units. Specifically the Multiple Host Robotic Architecture defines a set of constructs that ensure mobile platforms and devices within a platform are able to communicate between each other. Using this architecture, programmers are able to code to control the mobile platforms as well as simulate the entities in a virtual environment using the MHRA set of constructs.

Currently, SPAWAR is actively engaged in developing live and simulated behaviors in a robotic system or a system of systems to engage in mine countermeasures. The constructive simulation is a collaborative effort involving SPAWAR, DARPA, INEEL, and the US Army to use a mix of aerial and ground vehicles to search, detect, mark and possibly neutralize land mines.

The simulation consists of the Organic Air Vehicle (OAV), and the Unmanned Ground Vehicle (UGV). The OAV, a small, unmanned oscillating flight vehicle, is tasked to do a high speed scan of an area of probable land mines. The OAV would mark the location of land mines obtained at a low resolution using sensors capable of detecting land mines. Following the OAV, the UGV would seek out areas that are marked by the OAV and determine if the marked area is an actual mine with a higher resolution of sensors. The goal of the composite system is to detect the mines and reduce the number of false negatives, or the number of mines discovered by the UGV that were not discovered by the OAV. The next step in the process is to possibly neutralize the land mine using methods that would normally be employed by a human operator. The ground breaking nature of this collaborative effort is the push to have the robotic units perform the previously mentioned behaviors in an autonomous fashion.

Currently the two platforms tasked to take some undertaking is a six inch diameter OAV and the All-Terrain Robotic Vehicle (ATRV) to execute the aerial search and ground search in the simulation and live simulation.

E. AGENT SYSTEMS

An agent is a computer system that is situated and interacts with its environment. The relationship between the agent and its environment encompasses a cause and effect interaction. Since the agent is situated in the environment, any action the agent takes has an inherent effect on the environment. If the agent moves from its current location to a new location, the overall effect on the environment is that the previous position is no longer occupied and the current position of the agent is occupied. Albeit a trivial example, the environment has had some changed that can be sensed by other objects.

Figure 3. Agent Environment Interaction

Agent interactions with the environment is accomplished through tight coupling of the sensory input streams received through various methods and the action output streams used to affect the environment. The agent can update its internal belief about the environment and deliberate its next action or set of actions to try and affect the environment. The deliberation process conducted by the agent is driven by the agent's desires and intentions.

Agents embody their own intentions and goals. Agents may also have multiple goals and some may be in conflict at any given time. The agent determines which goals are active, or have higher priority, by use of a utility function. The utility function looks at the goal structure of the agent and determines what goals have higher priority and suggests to the agents the order of completion and resolves conflicts between competing goals. The utility function will be explained in more detail in chapter four as part of the implementation of the Agent-Economy.

THIS PAGE INTENTIONALLY LEFT BLANK

III. AGENT ARCHITECTURE

A. THE SINGLE AUTONOMOUS ROBOT: BENDER

1. Construction and Design of Bender

The goal of this project was to create a controlling architecture for a physical robot placed in a simple environment and implement the architecture to gain useful information for a constructing a simulation. The robot, Bender, is a Lemming, tracked wheel vehicle modified with an electronics cabinet, which houses the onboard processor, Global Positioning Satellite (GPS) receiver, sonar sensors, and a magnetic compass receiver. The high-level control involves: check alignment all the way through

- Navigating to a GPS waypoint
- Sense objects within its environment
- Avoid collision with obstacles in the environment

Bender was designed with the idea of adding more levels of architecture while not affecting the current levels of control. Bender's controlling architecture, written in Java, is responsible for processing information received by the physical stimuli experienced by the robot and control signals are sent back to the robot via a wireless Ethernet connection.

Figure 4. Bender Block Diagram

Architecturally, Bender is equipped with a BL2000 *Wildcat* coprocessor that is responsible for signal processing of raw data received from the GPS receiver, the magnetic compass and sonar sensors. The information is converted from analog and is sent to the Java program, running on a laptop via the wireless Ethernet connection. The BL2000 also handles low level processing of the motor controller by sending inputs received from the Java program to control basic maneuvers of left, right, forward, reverse and stop functions of the robot. When coupled with the Java program Figure 5, Bender is able to move according to the agent architecture and exhibit rational movements based on the current state of the environment, refer to Appendix A. Bender's movements are based on the current belief about the environment to include what objects are within sensing range, its current course, and distance from the next waypoint. Using this information, the java program sends a signal to the robotic platform to execute the next move to either avoid collision or continue moving towards the waypoint.

Figure 5. Bender Control Program

Figure 6. Bender Environment Interaction

2. Agent-Based Architecture of Bender

The Java program that is the "brain" of the robot is responsible for interpreting inputs received from the BL2000 and making decisions based on what is perceived as the environment. Perception is based on information received from the onboard sensors, specifically the GPS and sonar sensors. The perceived environment is based on stimuli that represent objects. The objective, or goal, of the agent is to get to waypoints that are stored internally or received from a controlling system. Waypoints can be received by a human operator manually entering the data, or generated by another robot in a distributed system when two or more robots are cooperating to arrive at a common goal. The single robot continues on its path in Subsumption type fashion as defined by Brooks (Brooks, 1986).

The most basic and primitive layer of control for Bender is to *cruise*. When Bender is in *cruise* mode, the assumption made is that no other modes are suppressing the agent's basic desire to move forward. The next level of control is to navigate to a specific waypoint. While the agent is moving forward and making progress towards the waypoint, within a certain threshold left or right of the desired course to get to the

waypoint, the waypoint controller relinquishes control to the basic *cruise* mode. Once the threshold to navigate to the waypoint is exceeded, the *correct* module suppresses the agents desire to move forward to turn and come to a new course that leads to the goal of getting to the desired waypoint in the *cruise* mode.

Figure 7. Module Selection Schema

The third module implemented in Bender is the *avoid* module. This module takes precedence and suppresses all lower modules to avoid collision with obstacles in its path. The obstacles can be stationary or dynamic objects. Stationary objects are things such as blocks, buildings, or any other stationary objects that are represented by a change in distance while bender is in a moving state. Dynamic objects are objects that have decreasing distance stimuli while Bender is in a moving state or a stationary state. When Bender is in a *cruise* mode and encounters an object in its path the *avoid* module takes control and attempts to come to a new heading that satisfies the goal of avoiding collision. Once the goal has been met, control is relinquished back to the *cruise* module. To prevent continual searching when control is relinquished from the *avoid* module, a minimum time threshold is implemented before the *correct* module can suppress the

cruise module after a collision avoidance maneuver is made, which would cause the robot to lock itself in front of large stationary objects disabling it from making its next goal.

B. MULTIPLE ROBOTS

1. Multiple Robots Using the Bender Architecture

The architecture for Bender scales up to two or more robots using the same techniques for the single robot. As the robots make their way towards their waypoints, they each sense their environment moving in a fashion as stated before by avoiding obstacles and resuming the proper course to reach their goal. The robots also sense other robots and avoid colliding with each other. The thing to note here is that there is no coordination involve with the robots at this point, however the low level goals for each individual robot is intact to move towards coordinated behavior.

By providing the robots with a common goal, and implementing the coordination mechanism, the robots can move towards behaviors that exhibit intelligence while accomplishing the common goal. The common goal that the robots attempt to accomplish is movement towards specific areas of interest designated as waypoints. When a robot has reached an area of interest or a waypoint, that information about the robots location can be submitted to a single robot or multiple robots to inform the collective that the goal for that waypoint has been met. The other robots can dismiss this goal since it has been accomplished and move towards movement to other waypoints.

The Agent-Economy presented in this thesis provides architecture for evaluating how robotic systems can be configured, in a simple environment, to minimize the number of robots used and the time it takes to find objects, while maximizing the certainty of finding the objects. This simulation is an attempt at finding the best configuration in simulation so that the information can be used to produce an agreeable solution in the real world as to the number of robots to use.

2. The Agent-Economy Architecture

As stated above, the Agent-Economy is focused on simulating different configurations of robotic units. The economy is an observable self-organizing system that takes into account the goals that are active and the resources to accomplish the goals. Knowledge about the environment is maintained locally, by each individual agent, basedon their own perception. Knowledge about the goals to be accomplished can be

maintained by a central controller, which can be a human controller, an agent or a number of agents that are responsible for limited amount of goals. Each individual agent has only local knowledge about the goal it is trying to accomplish while the central controller has knowledge about the goals and the resources to accomplish those goals. The system is similar to many organizations that have hierarchical control where lower level agents have specific knowledge about a small subset of a larger organizational goal. This is a layered approach to solving the overall common goal of the system.

Figure 8. Agent-based Layered Control

The layered control architecture is depicted in Figure 8, showing the individual agents responsible for movement towards their goal in the agent layer and the central controllers in the control layer that are responsible for resource management of the agents in the agent layer. The agents in the agent layer have local perspective and sense their environment in a local coordinate system to accomplish the goals that are received from the controllers in the control layer. As the agents accomplish the goals, they report back to the central controllers and move to the next goal in their goal structure or wait to receive new goals from the central controllers. Agents are aware of other agents as they sense them in their environments or receive information from the central controllers about other agents that are outside of their sensing range. Central controllers can query agents for status of local resources and the status of meeting their goals. If agents are

unable to meet their goals due to limited resources, the controller can arbitrate whether or not to keep the agent's resources working towards the current goal or allocate other agents to assume responsibility for meeting the goal.

The simulation discussed in Chapter IV deals primarily with the agent layer, however it can be scaled to include the architecture of the control layer. On the other hand, the architecture of the Bender project used both the control layer and the agent layer. Since Bender was actually situated in the real world, the agent layered involved Bender interacting with real world objects and sensing using GPS and sonar sensors. The control layer consisted of the Java program that was responsible for arbitrating the best possible route to get to the next waypoint. The task of the controller and the agent are combined into one layer in the *Agent Economy*.

In this layered approach it is important to note that the central controllers in the control layer are agents as well with a different goal structure than the agents in the agent layer. The central controllers also have the ability to sense other controller agents that are within their sensing range and possibly sense the agents that are under the control of the other controller agents. Goal sharing between controller agents is essential to the coordination efforts of the systems. If one controller has goals that can be accomplished by negotiating with another controller, the agent can coordinate to swap goals or relinquish responsibility to the other controller. The controller assigns a confidence level for the new controller in its ability to accomplish the goal and if the confidence level meets a certain threshold, the goal is swapped or relinquished to the next controller. The confidence level is a utility function that takes into account the goal and the resources the controller has to accomplish this goal.

C. SCALING THE ARCHITECTURE TO A REAL SCENARIO

In this section I will discuss an example of how the agent-based architecture can be used to simulate a real world scenario. By simulating the robotic units in a scenario before hand, the robotics engineer can determine key factors about the complete system. Things such as number of robots to use at the agent level can be realized as well as the number of controlling agents to use in the control level. The analysis that supports

arriving at the configuration will be discussed in Chapter V. Keep in mind that the *Agent Economy* simulation is a generic example as to how simulation can be used to help shape the real world robotic system.

1. Mine Counter-Measures Using Agent-Based Simulation

The scenario is based on using robots to conduct mine counter-measures. The scenario is envisioned by SPAWAR is to use a mix of autonomous ground and air vehicles to search, detect, isolate and possibly neutralize land mines in order to provide safe passage for an Army ground unit. The heterogeneous mix of robots would enable this dangerous task to be accomplished while relieving the soldier from entering into the dangers of a mine-field. The scenario uses autonomous air vehicles to search an area of interest and does high level scanning for possible mines and mark the position of all potential mines. Autonomous ground vehicles are then used to do a more detailed search of the positions given by the air vehicles and positively identify each position as a mine or a false detection. Once the positive mines are identified and marked, they can be isolated or neutralized.

The scenario presented here is concerned with movement of robotic units to the assumed positions of a mine. When a number of ground vehicles are deployed, the agents are to coordinate and organize themselves to arrive at an efficient solution for reaching the mines. The efficient solution should take into account the time it takes to interrogate all possible mines, and the resources each vehicle has to arrive at the goal of finding the mines. The self-organizing nature of the system relies on the fact that the vehicles are autonomously working at the agent layer, providing feedback to the central controllers in the control layer. As positions of the agents are updated and goals are met, the central controllers receive feedback from the agents and in turn update the agent's goal structure until all goals are met. The system is a semi-closed loop system since agents and controllers conduct business until all goals are met or until intervention by a human controller. By executing this system, the first goal of the mine counter-measure behavior to detect mines can be met.

2. Behaviors of the Ground Vehicles

The ground vehicles in the mine counter-measure scenario are responsible for interrogating locations of possible mines. The location of a possible mine given to the

ground vehicle agent is received from a controlling agent based on the parameters set forth by the needs of the controlling agent and the capabilities of the ground vehicle. Locations are generated by aerial vehicles that locate possible mines by scanning an area at relatively high speeds at a low resolution. The low resolution implies that there is a level of improbability with actually marking an actual mine with the idea of having false-positive identification of mines. Relating this to the Agent-Economy, ground behaviors that are specifically coded fall into three categories, avoid, navigate, and search.

a. Avoid

The most basic function or goal of a ground vehicle agent is to avoid obstacles in its path that it may possibly collide with. Objects to collide with are the other agents in its sensory range or fixed objects that it must navigate around. Using Brooks' subsumption architecture the avoid goal takes precedence over all other goals in order to preserve the integrity of the agent. If the agent continually collides with other objects, its resources severely degrade and it eventually is unable to complete goals that it is trying to accomplish. When looking at this from the point of view of a real robotic system, numerous collisions can severely degrade the operation of the system and require increased amount of maintenance and service in order to have the system function according to its goal structure. So the most basic behavior of the agent in this simulation is to avoid obstacles that it comes into contact with.

b. Navigate

The next important goal of the ground vehicle agent is to navigate. As demonstrated by Bender, navigation consisted of moving to a GPS location that is either implemented in software or received by a secondary controller. The secondary controller in the Agent-Economy simulation resides in the agent layer as one package. Navigation for the ground robot plays an important role in formulating proper paths to reach a goal. Often the robot must deviate from its intended course to negotiate obstacles that are in its path. After the maneuver has been made, the robot may find that it has strayed in a direction farther away from it goal. In this case the robot must take corrective actions to move towards its goal. A fitness function may be used to determine how well the robot is meeting its goal of navigating to an assigned waypoint. The fitness function will be described in more detail in Chapter IV.

3. Simulating the Scenario

The simulation for this scenario can be an important tool. One question that must be answered is how many robots are necessary to do such a task as mine-countermeasures. Since robotic systems can be expensive to build, the simulation could provide valuable information as to the starting point for building the robotic units. After the simulation is run enough to get a comfortable understanding of the problem, production can begin on building the number of suggested robots. Next comes testing in the actual environment. Testing in the environment is crucial to the usefulness of the simulation. Feedback is gained from testing in the environment as to how well the simulation works for providing insight to the real system. The simulation can then be tweaked to reflect the phenomena experienced in the live testing that was not initially accounted for in the simulation.

IV. IMPLEMENTATION

A. INTRODUCTION

This chapter discusses the implementation of the Agent-Economy using a simple environment constructed in Java. All files were coded and compiled using Borland's JBuilder 7 Enterprise edition, however the source code can be compiled using any computer with the Java Virtual Machine installed. The remainder of this chapter is a discussion of the classes involved in running the Agent-Economy.

B. ENVIRONMENT

The environment for the Agent Economy is a class developed in Java for the purpose of setting up agents that can maneuver and interrogate a simple world. The Environment class extends the Japplet class and implements the Runnable and Data interfaces. The Runnable interface enables the Environment class to implement Threads so that multiple threads can be executed in a time slicing fashion. The class contains an agentList that keeps track of all agents in the environment while the simulation is running. Each agent in the agentList has their own goal structure that is continually updated based on their utility function, see code in Appendix A. Once the Environment class is instantiated the init function is executed so that the threads may run. Inside the init function the environment is created without robots and setup to delineate the boundaries of the world and any objects that are situated in the world. Once the world data structures have been created the 2D graphics of the world are drawn to screen as well as the Graphical User Interface Components (GUI) that the user is able to manipulate while the simulation is running as shown in Figure 9.

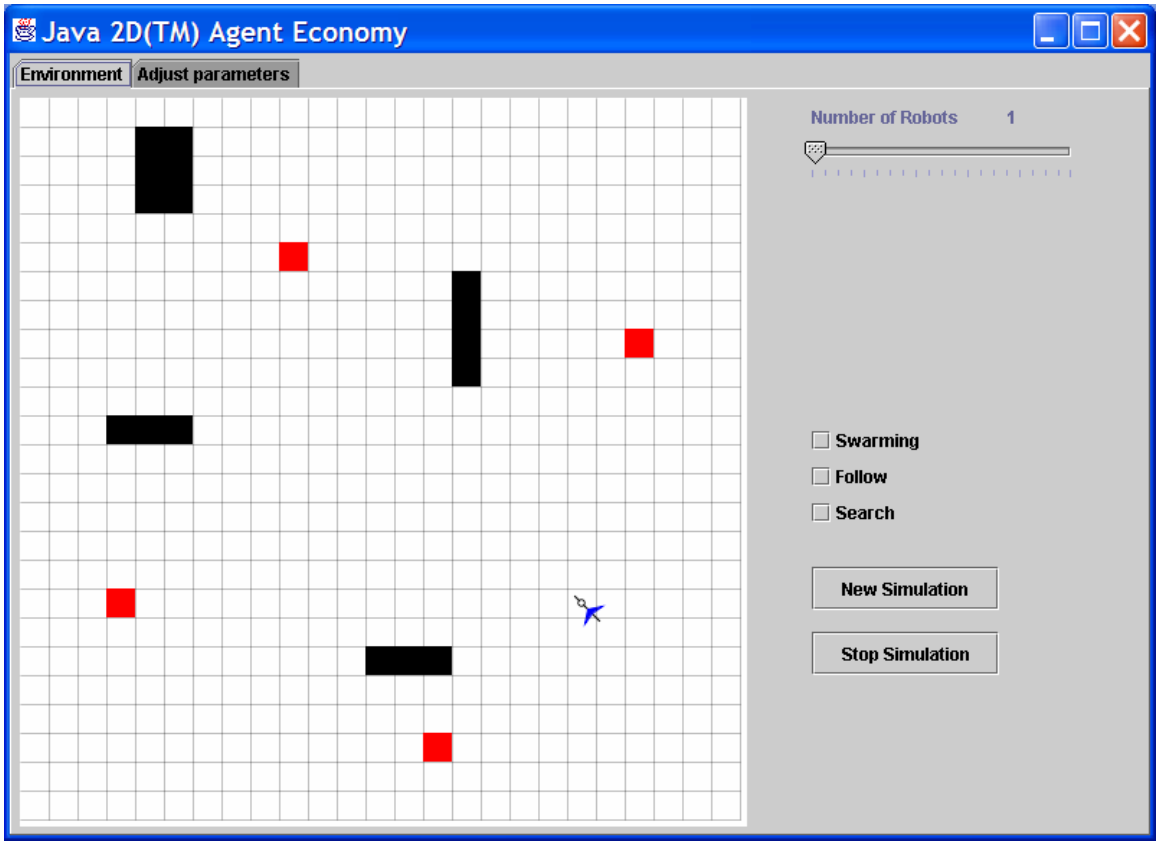

Figure 9. Agent Economy Screenshot

The objects in the world, represented by the black basic shapes, are primitive objects meant to simulate the presence of real world objects and are listed in the collision list of the environment. Waypoints are represented by red squares placed in the environment. The collision list is a simple array of locations in the environment to track collisions and location of agents, objects and waypoints that are situated in the environment. The collision array is the maintenance area for the boundaries of the world and data representation of all static objects in the environment. A value of zero represents an area that is not occupied by an agent or an object, and any value other that zero represents the presence of an object or and agent.

Agents are represented as a chevron with a directional arrow placed in the middle of the chevron. The GUI for this environment enables the user to change the number of agents that are situated in the environment and change the behavior that the agents exhibit. This is accomplished using a slider for the total number of agents and check boxes for the type of behavior that the agents exhibit. However, the functionality for the

behavior checkboxes was not implemented as part of this thesis. The GUI also has a tabbed pane to look at and adjust the system parameters. The functionality for adjusting the parameters during the simulation run were not implemented as part of this thesis as well. The window that appears once the simulation is in run mode is typical of windows displayed in a MS Windows ™ environment. Closing the window causes the simulation to terminate.

While each agent's position and next move is determined by their own goal structure, the drawDemo function is responsible for initiating the next move of the agents. Within this function each agent determines the next best possible move in order to avoid colliding with other agents or objects in the environment. The call to this function is made during each time step and runs until the user closes the window and terminates the simulation.

Collisions are calculated after each agent has made their move and a color coded scheme is used to notify the user when a collision has been made or an agent is endanger of colliding. While the agents are in no danger of colliding with another agent or object, their color is blue, once the possibility of colliding with something is apparent the agents turn yellow, and a collision causes the agents to turn red. The color coded scheme is simply used to notify the user of the event that a collision has occurred or may occur.

C. SITUATED AGENTS

The agents in the simulation are situated in a simple environment so that they may sense and detect other objects and agents while constructing there own local perspectives about their surroundings. The local perspective is the basis of their beliefs about the world. If an agent encounters no objects or agents during its sensing cycle, then that agent has a belief that it is the only thing populating the world. Sensing is accomplished by querying the environment based on a sensing distance of three grid units around the agents center similar to a sonar or an infrared sensor placed around a robot. Sensing by communicating with other agents can increase the agents overall sensing range and likewise the beliefs the agent has about the environment.

When an agent receives a sensory input through communication from another agent, the information can be used in the utility function with a certain level of trust. If

the information is fully trusted, then the agent's local perspective about environment is increased by a scale factor of one and the range of agent is extended to the distance of what the agent has received. For instance, if an agent has a sensing range of two units around its center and another agent five units out provides information about its local perspective with a range of two units around its center, the receiving agent assigns a confidence level between zero and one of the information received. If the confidence level is one then the agent increases its sensing range for that sensing cycle to five with a radius of two in the direction of the transmitting agent, but not to a distance of five around the receiving agents center since a complete radius gain of information would be out of the transmitting agents sensing range. This idea is especially important for the controlling agents since their local perspective is an aggregate of local perspectives from the agents that are situated in the environment. This concept will be further demonstrated in the simulation and discussed under the utility functions.

D. AGENT BASED GOAL STRUCTURES

Each agent has a goal structure that governs: (1) how they interact in the environment and (2) the behavior of each agent. A goal represents a task or state that the agent is trying to achieve. Agents may have multiple goals and work to accomplish each one individually or concurrently with other goals. An agent contains a goal list that ranks each goal by order of precedence. Agents attempt to accomplish higher precedence goals during each iteration and then move on to the next goal.

Goals in the agents are dynamic, in that during one cycle of moving towards completing a goal the agent may deem it necessary to make a goal that has lower precedence during the previous cycle to have a higher precedence during the current or next cycle. This decision is based on the utility function. An example of this behavior is especially important during collision avoidance. If an agent's primary goal is to navigate to a location it will have a higher precedence than the avoid collision goal so long as there are no objects or agents near that may cause a potential collision. However, as the agent moves throughout the environment, when it encounters a possible collision, the avoid collision goal then receives higher priority and the agent attempts first to satisfy this goal. While the simulation is running this restructuring of the goal list is transparent to the user. In fact agents are capable of accomplishing goals simultaneously so long as

the goals are not in conflict. An agent may very well avoid collision and navigate to a position at the same time, but many cases arise where avoiding collision can maneuver an agent to a position that is far from accomplishing the agents goal to navigate to a position. Determining which goals are active is based on the utility function of each agent and each goal is given a priority number.

E. UTILITY FUNCTIONS

The utility function is the brain of each agent. Within the utility function, the agent takes into account its current knowledge about the environment, its active goals and the resources that it has to accomplish each goal and determines its next best move in accomplishing the goals that are active. The utility function is formula based and produces a marker for the agent's next move to make during the next iterative cycle of the simulation.

In the Agent Economy simulation, the function findNextMove is the utility function responsible for movement of the agents. The function looks at the current position of the agent and all possible locations which the agent can move to. The agents attempt to move towards their goal while also avoiding collision with objects or other agents. The environment is marked with integers that indicate whether or not a position is filled. If a position contains a one, the position is filled by either an another agent or a static object while positions filled with a zero are empty and the agent accepts this position as a possible position to move to. The agents query the environment for the data of positions three units in all directions to simulate input from onboard sensors. When the agents receive information on the objects they are searching for within their sensing range, they set a priority level in the direction of the goal. Positions that are closer to the agent's goal and empty have a higher priority than positions that are either filled or farther away from the agent's goal.

Conflicts with each agent's goal may arise when two or more agents attempt to occupy the same position. When this occurs, the user is notified by the color coded scheme of the simulation. Since all agents make their decision to move based on their own local perspective, the simulation is set up so that each agent determines their next move based on the current location of every object and agent in the simulation. Once the decision of each agent's move has been made, the screen is repainted with each agent's

new position. If two agents move to the same position, a collision occurs and the color scheme reflects the occurrence of the collision. The next cycle of the utility function takes this into account and compensates to avoid further collision.

F. THE FITNESS FUNCTION

Each agent has to be able to determine how well it is meeting its goal in order to self correct itself to meet the goal. This is an important factor in the self-organizing economy. If an agent has a goal to get to a waypoint, the agent must know how well it's doing in accomplishing that goal. A fitness function is used to calculate or measure the effectiveness of its actions in accomplishing its goals.

The agent must be able to determine three things about the goals it is trying to accomplish.

1. The agent must know whether or not the goal is complete. If an agent has completed the goal there is no need to continue to try and complete the goal.
2. The agent should know whether or not it is capable of completing the goal. Again, if the agent is unable to complete a goal there is no real need for the agent to continue wasting resources to try and complete an unattainable goal.
3. Lastly, the agent should be aware if its actions are moving it closer to completing a goal.

This is the self-organizing mechanism of the agent that brings it closer to accomplishing the active goals in its goal structure. If an agent's goal is to travel to a waypoint north of its current location and all of the agent's moves are heading south of its current location, then the agent should be able to recognize this fact and take corrective actions to move closer to completing its goal of moving to the North located waypoint.

A fitness function is simply a method of measurement for answering the three important questions about the agents active goal, is my goal complete, can it be completed, and how are my actions in completing the active goal. Once these questions are answered in the fitness function, the agent can either remove the goal from its goal list because it is complete, abandon the goal because it is improbable to complete, or take corrective action to move closer to completing the goal. This is the basis of the fitness function.

G. REAL WORLD SCALING OF THE SIMULATION

The simulation created for this thesis uses a dimensionless scale for searching the environment. The agent's movements are in grid units and have no tangible meaning when used for real robots. Since there are a number of factors contributing to a robot's course and speed such as the type of surface, the force exerted by the robot and battery life, the simulation would need to be tailored to scale directly to an actual robotic system and is reserved for future work. The agents in the simulation move based on a random course, constant speed scale and have the luxury of doing so as long as the simulation is running. When using this type of simulation for actual implementation, all factors of the robots movement must be taken into consideration.

THIS PAGE INTENTIONALLY LEFT BLANK

V. RESULTS AND ANALYSIS

A. INTRODUCTION

This chapter describes the process of analysis for the Agent Economy simulation. The goal of the simulation is to get agents to various locations of the grid where potential items of interest were placed using the best mix of agents for the number of items in the environment. In doing so, the simulation mimics what an actual robotic system may encounter when allocating resources to move to each area of interest. The testing phase consisted of using one agent to interrogate the environment as a baseline and then scaling up to multiple agents interrogating the environment. Each run of the simulation is based on the time it takes the agents to find the items placed throughout the environment.

B. THE SINGLE AGENT TIME TRIAL.

During the run of the single agent, one agent is placed in the simulation at a random position with four static items of interest placed in the environment. The agent's goal is to navigate through the environment and come in contact with each item. The color-coding scheme of the agent alerts the user when contact has been made with the item. As soon as the simulation is running, the clock starts for the agents. A grid is placed over the environment to decrease the frame rate of the simulation to slow down the speed of the agent in order for the user to manageably time the agent's run. Figure 10 is a screen shot of the scenario featuring one agent searching for the items in the environment.

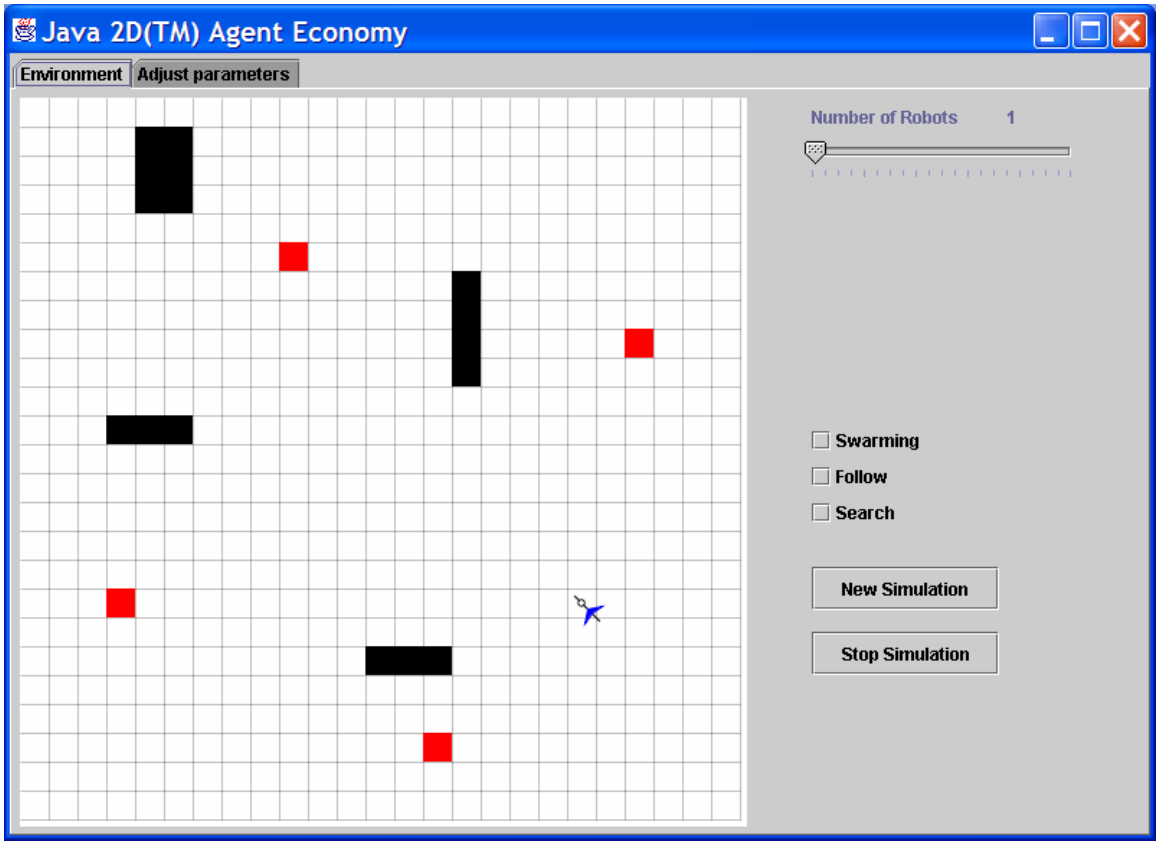

Figure 10. Single Agent Simulation Screenshot

C. MULTIPLE AGENT TIME TRIALS

The multiple agent time trials are conducted in a similar fashion as the single agent time trial using more agents to conduct the search of the environment. The same number of items are placed in the environment and the agent's start the simulation at random positions. The goal here is to determine the proper number of robots necessary to optimize the time required to find the items and to reduce the inefficient use of too many agents. Certainly the agents can reduce the amount of time it takes to find the items if the environment is densely populated with agents, however many agents would be in the environment wandering around with no real value as the other agents find the items placed throughout the environment. Given this fact, the key is to find, on average, the proper number of agents necessary to locate each item. Figure 11 below is a screen shot of multiple agents in the environment searching for the items of interest.

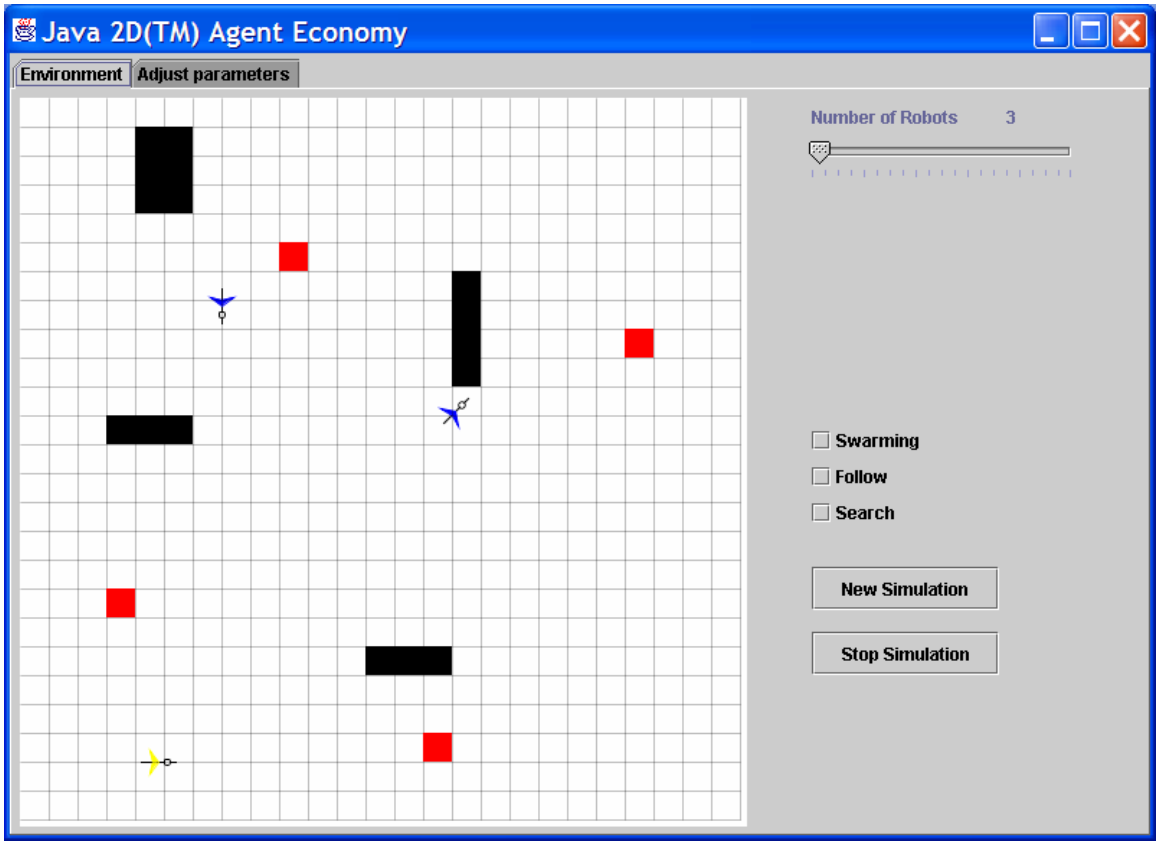

Figure 11. Multiple Agent Simulation Screenshot

D. RESULTS

The simulation was run using one, two, three, and four agents searching the environment for four items placed statically amongst the agents and the other objects. The simulation was run ten times for each group of agents and they were timed based on how quickly they were able to find the items. A maximum time limit was set at two minutes or one-hundred twenty seconds. The assumption for all groups was that the agents should be able to find the objects with in 2 minutes and they're score was set to two minutes if all items were not found in this time period. The results are shown below all times are in seconds with a maximum of 120:

Run #	1 Agent	2 Agents	3 Agents	4 Agents
1	**120**	**120**	**31**	**35**
2	**44**	**43**	**23**	**23**
3	**78**	**91**	**54**	**24**
4	**45**	**39**	**35**	**45**
5	**39**	**120**	**105**	**39**
6	**55**	**120**	**29**	**29**
7	**59**	**120**	**33**	**38**
8	**120**	**48**	**21**	**28**
9	**51**	**41**	**19**	**26**
10	**79**	**70**	**21**	**87**

Table 1. Simulation Raw Data

The raw data for each agent was then processed to get the best and worst scores, the overall average of the agent system, and the number of failed attempts by each group of agents. Note also that the data does not reflect interaction and cooperation among agents, the data is purely numerical based on the time it took the agents to complete the search of the environment. Table 2 shows the overall performance of each group of agents in the scenario.

# Agents	Fastest Time	Slowest Time	Avg. Time	# Fail Search
1	**39**	**Max**	**69**	**2**
2	**39**	**Max**	**81**	**4**
3	**19**	**105**	**37**	**0**
4	**23**	**87**	**37**	**0**
5 Saturation Point, too many agents to interrogate the environment efficiently.				

Table 2. Simulation Completion Times

Using the raw data generated from the simulation runs a Figure of Merit (FOM) is computed to see which agent configuration would be best suited for this scenario. The FOM takes into account three variables to minimize: Equation Section 5

1. Number of agents used
2. Average time to complete the search
3. Number of failed attempts (certainty of searching the environment)

We want to maximize the FOM. What we control in the equation is the number of robots/agents per run. This selection drives the "average time to search" and "number of failed" searches parameters and ultimately, the FOM. The following equation applies:

$$\mathrm{FOM} = c_1(\alpha) + c_2(\beta) + c_3(\gamma) \tag{5.1}$$

- α = *average time of search*
- β = *number of robots/agents*
- γ = *number of failed searches*

For the purposes of this simulation the constants used in the formula where weighted based on the importance of each variable. Since the goal is to complete the search of the environment, the constant for failed searches was penalized a bit more than the other variables. The cost of adding more agents to the environment is extremely inexpensive in terms of performance degradation so the constant doesn't bear a heavy

penalty; however when actually building a expensive robotic system the penalty should be somewhat stiffer than what this simulation renders. Finally, the average time of search incurs a slightly heavier penalty in this simulation since the goal is based on searching the environment. The formula for this simulation is as follows:

$$\text{FOM} = -\left[0.3(\alpha/10) + 0.2(\beta) + 0.5(\gamma)\right] \tag{5.2}$$

The results of the simulations using the figure of merit formula revealed that the optimal number of robots for the simulation was to use three agents to search the environment. Figure 3 illustrates the results of each agent configuration figure of merit scores.

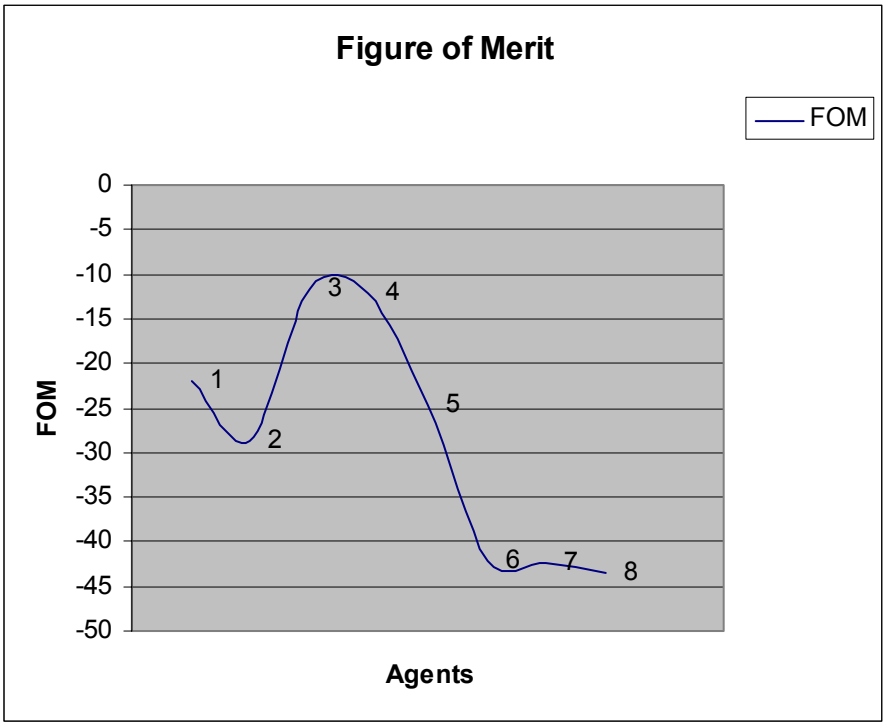

Figure 12. Figure of Merit Results

The simulation shows that there is a significant decrease in the figure of merit score when using two agents instead of one. The figure of merit then increases significantly when using three agents primarily due to the fact that there were no failed attempts and the average completion time was significantly reduced. Since both the three

and four agent simulation produced the same average completion time with no failed attempts, the difference in figure of merit score relied on difference in number of agents.

The scenario was also run using five to eight agents in the environment. What was discovered is that five or more agents saturated the environment, and the agents performed poorly; see appendix for results. Five or more agents tended to clutter areas and essentially get in each agents way when trying to navigate to an item of interest and in many cases timed out due to so much confusion amongst the agents.

E. CONCLUSION

Running the simulation is not conclusive of exactly how many real robotic systems should be used when conducting a search for items in an area; however it does give an indication of possible starting points for the robotics engineer. After conducting as simulation such as the Agent Economy, the work can begin on putting together a team of robotic systems and testing whether or not the simulation produces fairly accurate suggestions for putting real robotic systems in play.

Using software simulation and agent based control structures is a good starting point for building actual robotic systems. The guidelines set forth by the simulation can provide a glimpse as to how an actual system may operate. After testing out theories of the simulation in the real world, the simulation can then be revised to more accurately reflect some of the real world phenomena not initially accounted for in the simulation. The relationship is circular in that simulation can drive some of the decisions of the real system and the real systems can suggest modifications to be made in simulation. Eventually both systems can move closer to a predictive and insightful relationship.

THIS PAGE INTENTIONALLY LEFT BLANK

VI. CONCLUSIONS AND FUTURE WORK

A. INTRODUCTION

This thesis has attempted to show the logical connection between agent-based software simulation and agent-based robotic systems. Implementing what can be done in software in a real system is not as cut and dry as some researchers would like to believe the process is, however by gaining an understanding of the real robotic system capabilities and needs, the software to produce behaviors capable in simulation can be realized.

B. SIMULATION AND REAL ROBOTICS

Simulating robotic systems and implementation of behaviors in a real robotic system should be a tightly coupled process in order to produce robotic systems that are useful in the real world. As we move closer to utilizing robotic systems, the value of simulation increases to reduce the number of errors and rework in designing and engineering behavior based robotic systems.

Once the nature and goals of a robotic system are defined, simulation is a great tool to explore different configurations of robots as well as the proper mix of heterogeneous robotic systems. Trial and error methods of finding the proper mix of robotics systems can be expensive and in some cases improbable when efforts are spent to design the system only to find that it is not the proper mix of robotic systems.

Tightly coupling simulation with the real robotic systems allows the designs of the robotic system to drive the proper course of the simulation and the results of the simulation can be used to tailor the design, engineering and employment of the robotic units. By using agent-based simulation large numbers of configurations can be explored at relatively low cost and in an expedient manner. Agents allow the system designer to view the possible configurations using small code segments that produce seemingly rich and complex behaviors.

C. FUTURE WORK

As with many thesis', the scope of the work to be accomplished narrowed over the course of writing and implementation. Many features and elements of the initial

conception have necessarily have been delayed in order to present a complete representation of the Agent Economy software with a functional code base. The following section describes some of the avenues for future work, as well as some thoughts on implementation and benefits the robotic researcher or simulation modeler might encounter.

1. Sensor Integration in Simulation

The problem with many robotic simulations, Agent Economy included, is the lack of sensor representations in the simulation. The Agent Economy forewent the use of sensors based on constraints that were modeled in the Bender project. Using these constraints, the simulation made no attempt at representing information that would not be represented using the Bender robot. Quite often simulations provide the advantage of querying the environment using listeners that are not available in any form or fashion the real robotic system. That being said, there is great benefit in modeling the types of sensors that the real robotic systems employ in order to gain a true representation of what may or may not work in the real system.

The simulation would benefit greatly from a sensor manager package that acts as a referee between the environment and the agents or robots that are in the environment. The sensor manager would look at the characteristics of the sensors employed by the robots and the objects that the sensors could possibly recognize. Careful consideration must be insured beyond this point as to how much of that information is processed by the robot. The beckoning question is whether or not the robot or the sensor manager is responsible for the modeling of processing that information. One argument is that the task be left to the robot and to ensure that the robot is equipped with a mechanism to allow for missed information. Just as no sensor is perfect in capturing all sensor information the sensors in the simulation miss information that is received from the sensor manager. The sensor manager could be responsible for sending sensor information to the sensor and model missed information, but the designer should not try and overload the sensor manager with trying to model all aspects of the sensor environment such as wave propagation.

2. Robotics Implementation

The goal of this thesis is to research the use of agent-based technologies for real world robotic applications. Although I would have liked to take the simulation a step further an implement the use of agent-based goal structures in a real robotic system and a team of coordinated robots, that work is beyond the scope of this thesis. Further research on robotic systems can take the ideas presented in this thesis to implement rational thinking agents in a robotic platform to negotiate their environment and coordinate their resources to accomplish a common goal.

An actual implementation of the agent-economy could consist of simple waypoint navigation with one controller agent orchestrating waypoints to one or more robots, while the agents or robots themselves make their way to the designated waypoints using their own local perspective and constructing a mental model of the environment in which they are situated. This can be further escalated by creating physical goals for the robots to accomplish and use the utility function to judge whether or not they have met the goals. The goals can be as simple as getting to the next waypoint or as complex as performing a design specific task such disabling a land mine if that resource and capability is available to the robot. The possibilities of the agent-based robot are limitless.

3. Integrating Simulation and Live Testing

Simulation and live testing and design of the robotic system should go hand in hand. The simulation should be used to drive decisions on design and the constraints of a live test at a relatively low cost. While the live testing should reveal real world phenomena that were not calculated by the simulation and can in turn be implemented to give a more accurate portrayal of what should happen once the real robots are utilized. Since robotic systems are expensive to build, operate and maintain, a sound objective for the total system would be to build one robot and interact with the simulation to work in conjunction with tens, hundreds even thousands of robots as if they were actually present in the environment. Once the simulation integrated with the live testing has produced a proper mix of robots, heterogeneous or homogeneous, the robotics research can focus on building the number of robots with the proper onboard systems that will satisfy the requirements of the complete system. System integration with simulation can be a powerful tool to use when designing robotic teams.

D. CONCLUSION

Much work has been done to gain insight from natural systems and to mimic their behaviors in software. The complex behaviors observed in these natural systems emerge from relatively simple rules of interaction. Realizing these simple interactions has allowed researchers to create complex adaptive systems using simple programming constructs by use of agent-based goal structures. While still in its infancy in the robotics community, the same structures devised to govern software agents can also be implemented in robotic systems to govern complex behaviors based on simple agent-based goals and interactions.

Significant room is left to improve upon in the Agent Economy simulation. Nevertheless, the system as presented provides a basis for using agent-based software for developing a roadmap for robotic systems that benefit from software simulation. It demonstrates that agents, although unpredictable at times, may exhibit useable behaviors and create a complex system of robotic units that cooperate to achieve a common goal or task that is defined by the system designer.

APPENDIX A. AGENT ECONOMY CODE

Appendix A is is a compilation of the code for the Agent Economy simulation. There are four classes in the simulation and one interface. The code is written in Java using JBuilder 7 Enterprise edition. What follows is each class in the following order:

- Environment
- Agents
- Robots
- GUI
- Data
- Raw data from the simulation runs

A. ENVIRONMENT

```
import java.awt.*;

import java.awt.event.*;

import javax.swing.*;

import java.awt.image.BufferedImage;

import java.awt.geom.*;

import java.util.*;

/**

 * Title: Agent Economy

 * Description: Simulation of Searching Robots

 * Copyright:    Copyright (c) 2003

 * Company: NPS

 * @author: Monty Williams

 * @version 1.0

 */

public class Environment extends JApplet implements Runnable, Data {
```

```
/**
 * index = 0 => speed 1 left turn
 * index = 1 => speed 2 left turn
 * index = 2 => speed 1 straight ahead
 * index = 3 => speed 2 straight ahead
 * index = 4 => speed 1 right turn
 * index = 5 => speed 2 right turn
 * index = 6 => speed 0
 */
static short wayPoints [] = { 0,0,0,0,0,0,0};
static int step = 0;
Thread thread;
static Vector agentList;
static {
   agentList = new Vector();
}
private BufferedImage bimg;
public Environment() {
   setBackground(Color.white);
   init();
}
public void init(){
   initWithoutRobots();
   for( int idx=0; idx < Data.numberOfXRobots ; ++idx ){
```

```
            int x = (int)(Math.random()* (Data.numberOfGrids - 8) ) + 4;
            int y = (int)(Math.random()* (Data.numberOfGrids - 8) ) + 4;
            agentList.add(new Robot((float)(Data.gridSize*x) , (float)(Data.gridSize* y),
Data.courses [idx % 8] ));
        }
        fillTheCollision();
    }
    public void initWithoutRobots(){
        for( int row=1; row < Data.numberOfGrids  ; ++row ){
            for( int column=1; column < Data.numberOfGrids  ; ++column ){
                Data.collision [ row ][ column ] = 0;
            }
        }
        for( int row= 0 ; row < Data.numberOfGrids  ; ++row ){
            // the outer frame is enclosed
            // right and left
            Data.collision [ Data.numberOfGrids - 1 ][ row ] = 1;
            Data.collision [ 0 ][ row ] = 1;
            //top and buttom
            Data.collision [ row ][ 0 ] = 1;
            Data.collision [ row ][ Data.numberOfGrids - 1 ] = 1;
        }

        /* Fill in the collision attributes of the upper leftmost primitive object.
```

The agents will sense these objects just as they sense the border of the envirionment and attempt to avoid colliding with these objects

*/

```
Data.collision [ 4 ][ 1 ] = 1;
Data.collision [ 4 ][ 2 ] = 1;
Data.collision [ 4 ][ 3 ] = 1;
Data.collision [ 5 ][ 1 ] = 1;
Data.collision [ 5 ][ 2 ] = 1;
Data.collision [ 5 ][ 3 ] = 1;

Data.collision [ 3 ][ 11 ] = 1;
Data.collision [ 4 ][ 11 ] = 1;
Data.collision [ 5 ][ 11 ] = 1;

Data.collision [ 15 ][ 6 ] = 1;
Data.collision [ 15 ][ 7 ] = 1;
Data.collision [ 15 ][ 8 ] = 1;
Data.collision [ 15 ][ 9 ] = 1;

Data.collision [ 12 ][ 19 ] = 1;
Data.collision [ 13 ][ 19 ] = 1;
Data.collision [ 14 ][ 19 ] = 1;
```

```
    Data.collision [ 3 ][ 17 ] = 1;
    Data.collision [ 9 ][ 5 ] = 1;
    Data.collision [ 14 ][ 22 ] = 1;
    Data.collision [ 21 ][ 8 ] = 1;
  }
  /**
   * Runnable interface's run method placed in this object so
   * we will use this run function in Thread() object.
   */
  public void start() {
    // The objects that implement the Runnable interface are passed into
    // the constructor for the thread object. When the thread starts
    // it will call the run() method of the object passed in.
    thread = new Thread(this);
    thread.setPriority(Thread.MIN_PRIORITY);
    thread.start();
  }
 /**
   * It is necessary to have a method or block of code executed by only
   * one thread at a time. The synchronized keyword is used to achieve that.
   * When close the simulation the thread status changes to null and edit the
   * program from the run()method while loop.
   */
  public synchronized void stop()
```

```
{
    thread = null;
}
/**
 * Runnable interface run method is overridden.In this method we call
 * repaint()
 *
 */
public void run() {
    Thread me = Thread.currentThread();
    while (thread == me) {
        repaint();
        try {
            thread.sleep(1);
        } catch (InterruptedException e) { break; }
    }
    thread = null;
}
/**
 * paint() will call this function for initialization and every rotation state.
 * This function will place the graphical objects to the frame.
 *
 * @param w  The width of dimension.
 * @param h  The height of dimension.
```

```
 * @param g2 Graphics2D type object.
 */
public void drawDemo(int w, int h, Graphics2D g2) {
   for( int idx=0; idx < Data.numberOfXRobots ; ++idx ){
      ((Robot)agentList.elementAt(idx)).drawRobot( g2);
   }
   if( step > 10 ){
      //findColors();
      findNextMove();
      //findColors();
      step = 0;
   }
   ++step;
}
/**
 * This function returns a graphic space with (w,h) dimension
 *
 *
 * @param w  The width of dimension.
 * @param h  The height of dimension.
 * @return  Graphics2D type object.
 */
public Graphics2D createGraphics2D(int w, int h) {
   Graphics2D g2 = null;
```

```
    if (bimg == null || bimg.getWidth() != w || bimg.getHeight() != h) {
        bimg = (BufferedImage) createImage(w, h);
        //reset(w, h);
    }//end if
    // Creates a Graphics2D, which can be used to draw into this
    // BufferedImage[ createGraphics() ]
    g2 = bimg.createGraphics();
    g2.setBackground(getBackground());
    g2.clearRect(0, 0, w, h);
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    return g2;
}
public void paint(Graphics g) {
    Dimension d = getSize();
    Graphics2D g2 = createGraphics2D(d.width, d.height);
    g2.setBackground(getBackground());
    g2.clearRect(0, 0, d.width, d.height);
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                            RenderingHints.VALUE_ANTIALIAS_ON);
    drawDemo(d.width, d.height, g2);
    drawGrid(g2);
    g2.fillRect(80,20,40,60);
    g2.fillRect(60,220,60,20);
```

```
        g2.fillRect(240,380,60,20);
        g2.fillRect(300,120,20,80);
        g2.setColor(Color.red);
        g2.fillRect(60, 340, 20, 20);
        g2.fillRect(180, 100, 20,20);
        g2.fillRect(280, 440, 20,20);
        g2.fillRect(420, 160, 20, 20);
        g2.setColor(Color.black);
        g2.dispose();
        g.drawImage(bimg, 0, 0, this);
    }
    public void drawGrid( Graphics2D g2){
        g2.setStroke(new BasicStroke(0.05f));
        int gSize = Data.gridSize;
        //Data.numberOfGrids = 550 / gSize;
        for (int i = 1; i <= Data.numberOfGrids; ++i){
          g2.draw(new Line2D.Float( 0.f, (float)i*gSize,
(float)Data.numberOfGrids*gSize, (float)i*gSize ));
          g2.draw(new Line2D.Float((float)i*gSize, 0.f, (float)i*gSize,
(float)Data.numberOfGrids*gSize ));
        }
        g2.setStroke(new BasicStroke(1.0f));
    }
    public void fillTheCollision(){
        for ( int listIndex = 0 ; listIndex < agentList.size(); ++listIndex){
```

```
            int x = Math.round(((Robot)agentList.elementAt(listIndex)).x/Data.gridSize);
            int y = Math.round(((Robot)agentList.elementAt(listIndex)).y/Data.gridSize);
            Data.collision [ x ][ y ] += 1;
        }
    }
    public void findNextMove(){
        for ( int listIndex = 0 ; listIndex < agentList.size(); ++listIndex){
            int x = Math.round( ((Robot)agentList.elementAt(listIndex)).x/Data.gridSize);
            int y = Math.round( ((Robot)agentList.elementAt(listIndex)).y/Data.gridSize);
            ((Robot)agentList.elementAt(listIndex)).x = (float)(x* Data.gridSize);
            ((Robot)agentList.elementAt(listIndex)).y = (float)(y* Data.gridSize);
            int head = (int) ((Robot)agentList.elementAt(listIndex)).heading;
            int speed = ((Robot)agentList.elementAt(listIndex)).speed;
            int index = 0;//wayPoints index
            int tempHead = -(head - 90);
            int degree = tempHead+45;
            if ( degree<0 ) degree+=360;
            else if( degree>360 ) degree-=360;

           // check the borders
            if ( x > 0 && y > 0 && x < Data.numberOfGrids -1 && y < Data.numberOfGrids -1) {
                //System.out.println("inner in borders" );
                    // inner square
```

```
            for (int i = 0 ; i < 3 ; ++i  ){
                if ( degree<0 ) degree+=360;
                else if( degree>360 ) degree-=360;
                //System.out.println(" degree = " + degree);
                int ix =(int) Math.round(Math.cos(Math.toRadians(degree)));
                int iy = (int)-Math.round(Math.sin(Math.toRadians(degree)));
                //System.out.println(" ix iy = " + ix + " " + iy);
                if (Data.collision [x + ix][y + iy] >= 1 ){
                    wayPoints [index] = -5  ;
                }
                index +=2;
                degree -= 45;
            }
        }
        else{
            for (int i = 0 ; i <= 4 ; i+=2  ){
                wayPoints[i] = - 10;
            }
        }
        // check the borders
        if ( x > 1 && y > 1 && x < Data.numberOfGrids -2 && y <
Data.numberOfGrids -2) {
            index = 1;//wayPoints index
            degree = tempHead + 45;
```

```
    // outer square
    for (int i = 0 ; i < 3 ; ++i  ){
        if ( degree<0 ) degree+=360;
        else if( degree>360 ) degree-=360;
        int ix = (int)(2*Math.round(Math.cos(Math.toRadians(degree))));
        int iy = (int)(-2*Math.round(Math.sin(Math.toRadians(degree))));
        wayPoints [index] = wayPoints [index-1];
        if (speed == 1 )
            wayPoints [index - 1] += 1  ;
        else if(speed == 2)
            wayPoints [ index ] += 1  ;
        if (Data.collision [x + ix][y + iy] >= 1 ){
            wayPoints [index] = -5  ;
        }
        index +=2;
        degree -= 45;
    }
}
else{

    for (int i = 1 ; i <= 5 ; i+=2  ){
        wayPoints[i] = - 10;
    }
}
```

```
        // Now for theCircle punishment
        if(((Robot)agentList.elementAt(listIndex)).circle[0] > 4 ){
           wayPoints[0]=(short)
(-((Robot)agentList.elementAt(listIndex)).circle[0]/2);
           wayPoints[1]                    =                    (short)(-
((Robot)agentList.elementAt(listIndex)).circle[0]/4);
           ((Robot)agentList.elementAt(listIndex)).circle[0]  = 3;
        }
        else{
           ++wayPoints[0];
           ++wayPoints[1];
        }
        if(((Robot)agentList.elementAt(listIndex)).circle[1] > 4 ){
           wayPoints[4]                    =                    (short)(-
((Robot)agentList.elementAt(listIndex)).circle[1]/2);
           wayPoints[5]                    =                    (short)(-
((Robot)agentList.elementAt(listIndex)).circle[1]/4);
           ((Robot)agentList.elementAt(listIndex)).circle[1]  = 3;
        }
        else{
           ++wayPoints[4];
           ++wayPoints[5];
        }
        for (int i = 0 ; i < 7 ; ++i  ){
        }
```

```
//Now find the best choice
int position = 0;
for (int i = 1 ; i < 7 ; ++i  ){
    if (wayPoints[position] < wayPoints[i] )
     position = i;
}
if (position == 0 || position == 1){
    //left turn
    ((Robot)agentList.elementAt(listIndex)).left = 1;
}
else if (position == 4 || position == 5){
    //left right
    ((Robot)agentList.elementAt(listIndex)).right = 1;
}
else if (position == 6 ){
    //stop and turn randomly
    int rnd = (int)(Math.random());
    if (rnd == 0)//left
        ((Robot)agentList.elementAt(listIndex)).left = 1;
    else//right
        ((Robot)agentList.elementAt(listIndex)).right = 1;
    ((Robot)agentList.elementAt(listIndex)).speed = 0;
}
if (position == 1 || position == 3 || position == 5){
```

```
            //speed 2
            ((Robot)agentList.elementAt(listIndex)).speed = 2;
        }
        else if (position == 0 || position == 2 || position == 4){
            //speed 1
            ((Robot)agentList.elementAt(listIndex)).speed = 1;
        }
        for (int i = 1 ; i < 7 ; ++i  ){
            //System.out.println( "wayPoints[" + i + " ]" + wayPoints[i]);
            wayPoints[i] = 0;
        }
    }//end for of list
    // clear the old collision positions
    initWithoutRobots();
    for ( int listIndex = 0 ; listIndex < agentList.size(); ++listIndex){
        ((Robot)agentList.elementAt(listIndex)).correctNewHeading();
    }
    findColors();
}//end findNextMove()
public void findColors(){
    for ( int listIndex = 0 ; listIndex < agentList.size(); ++listIndex){
        int x = Math.round( ((Robot)agentList.elementAt(listIndex)).x/Data.gridSize);
        int y = Math.round( ((Robot)agentList.elementAt(listIndex)).y/Data.gridSize);
        ((Robot)agentList.elementAt(listIndex)).status = (Color)ifCollided( x,y );
```

```
    }
  }
  /**
   *
   * @param TX  target cell x coordinate
   * @param TY  target cell y coordinate
   */
  public Color ifCollided(int TX, int TY)
  {
    int
      startRow,   // starting row
      endRow,    // ending row
      startCol,    // starting column
      endCol;     // ending column
      int ZERO = 0;   // for precondition check
    Color rV = Color.blue;
    // Preconditions: If TX or TY is outside array bounds
    // 0 - Data.numberOfGrids and 0 - Data.numberOfGrids, unprdictable results.
    if ( ( ZERO <= TX  && TX < Data.numberOfGrids ) && ( ZERO <= TY  &&
TY < Data.numberOfGrids ) ) {
      // create row-column start and end from TX and TY
      // for rows
      if ( TX == Data.numberOfGrids - 1){
        endRow = TX;
```

```
}//end if
else{
   endRow = TX + 1;
}//end else

if ( TX == ZERO ){
   startRow = TX;
}//end if
else{
   startRow = TX - 1;
}//end else

// for columns
if ( TY == Data.numberOfGrids - 1){
   endCol = TY;
}//end if
else{
   endCol = TY + 1;
}//end else
if ( TY == ZERO ){
   startCol = TY;
}//end if
else{
   startCol = TY - 1;
```

```
        }//end else
        // print neighbors here
        for (int arrayIndexRow = startRow; arrayIndexRow <= endRow ;
          arrayIndexRow++ ){
          for (int arrayIndexCol = startCol ; arrayIndexCol <= endCol ;
            arrayIndexCol++ ){
            if( (Data.collision [ arrayIndexRow ][ arrayIndexCol ] >= 1)&&
              (TX != arrayIndexRow) && ( TY != arrayIndexCol)){
              rV = Color.yellow;
            }// end of if
          }// end of for
        }//end of for
        if (  Data.collision[TX][TY] > 1 ) {
          //System.out.println("Red "+Data.collision[TX][TY]);
          rV = Color.red;
        }// end of if
      }// end of if
      //System.out.println(rV);
      return rV;
    }// end ()
  } // End Demo class
```

B. AGENTS

```
import java.awt.*;
import java.awt.event.*;
import java.awt.geom.*;
```

```
import javax.swing.*;
/**
 * Title: Agent Economy
 * Description: Simulation of Searching Robots
 * Copyright:    Copyright (c) 2003
 * Company: NPS
 * @author: Monty Williams
 * @version 1.0
 */
/**
 * The Rotate class renders rotated ellipses and includes controls for
 * choosing the increment and emphasis.  Emphasis is defined as which
 * ellipses have a darker color and thicker stroke.
 */
public class Agents extends JApplet {
   GUI gui;
   public void init() {
      gui = new GUI();
      getContentPane().add(gui);
   }

   public void start() {
      gui.demo.start();
   }
```

```
    public void stop() {
        gui.demo.stop();
    }
    public static void main(String s[]) {
        Agents demo = new Agents();
        demo.init();
        JFrame f = new JFrame("Java 2D(TM) Agent Economy");
        f.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {System.exit(0);}
        });
        f.getContentPane().add("Center", demo);
        f.pack();
        f.setSize(new Dimension(800,580));
        f.show();
        demo.start();
    }
} // End Agents class
```

C. ROBOTS

```
import java.util.*;
import java.awt.geom.*;
import java.awt.*;

/**
 * Title: Agent Economy
 * Description: Simulation of Searching Robots
```

```
 * Copyright:    Copyright (c) 2003
 * Company: NPS
 * @author: Monty Williams
 * @version 1.0
 */
public class Robot {
    static AffineTransform at ;
    static short step;
    float x ;
    float y ;
    float increment;
    Color status;

    short heading ;
    short type ;
    short speed;
    short left = 0;
    short ahead = 1; //default
    short right = 0;
    int circle[] = {0,0};

    public Robot(float x, float y, short heading )
    {
        this.x = x;
```

```
    this.y = y;
    this.heading = heading;
    type = 1;
    this.speed = (short)( Math.random() + 1 );
    status = Color.blue;
}
public void fillTheFutureCollsion(){
    if (this.heading < 0 ){
        this.heading += 360;
    }
    else if(this.heading > 360 ){
        this.heading -= 360;
    }
    // fill in the destination position as collision
    int degree = -(this.heading - 90);
    if ( degree<0 ) degree+=360;
    else if( degree>360 ) degree-=360;
    int cX = Math.round(this.x/Data.Data.gridSize);
    int cY = Math.round(this.y/Data.Data.gridSize);
    /*int fX = cX + (int)( speed* Math.round(Math.cos(Math.toRadians(degree))));
    int fY = cY + (int)(-speed* Math.round(Math.sin(Math.toRadians(degree))));*/
    int fX = cX;
    int fY = cY;
```

```
        if ( ( 0 <= fX  && fX < Data.Data.numberOfGrids ) && ( 0 <= fY  && fY <
Data.Data.numberOfGrids ) ) {
            Data.Data.collision [fX][fY] += 1;
        };
    }
    public void correctNewHeading(){
        if (this.left == 1){
            this.heading = (short) (heading - 45) ;
            this.left = 0;
            fillTheFutureCollsion();
            ++circle[0] ;
        }
        else if(this.right == 1){
          this.heading = (short)(heading + 45) ;
          this.right = 0;
          fillTheFutureCollsion();
          ++circle[1] ;
        }
        else {
            fillTheFutureCollsion();
        }
    }
    public void drawRobot( Graphics2D g2)
    {
```

```
//correctNewHeading();
int degree = -(heading - 90);
if ( degree<0 ) degree+=360;
else if( degree>360 ) degree-=360;
increment = (float)2.*speed ;
float ix = (float)(Math.cos(Math.toRadians(degree)) * increment);
float iy = (float)(-Math.sin(Math.toRadians(degree)) * increment);
this.x = this.x + ix ;
this.y = this.y + iy ;
at = AffineTransform.getRotateInstance(Math.toRadians( heading ),x, y);
g2.setStroke(new BasicStroke(1.0f));
//Robot's head diameter
float diameter = 4.f ;
g2.draw(at.createTransformedShape(new Line2D.Float(x, y-2*diameter, x, y -
diameter )));
//head
g2.draw(at.createTransformedShape(new Ellipse2D.Float(x-diameter/2, y-
diameter, diameter , diameter )));
g2.draw(at.createTransformedShape(new Line2D.Float( x, y, x, y  + 4*diameter
)));
// define the arrow path
GeneralPath wings = new GeneralPath();
wings.moveTo( x, y + diameter );
wings.lineTo( x - 2.f*diameter , y + 2.5f*diameter);
```

```
        wings.lineTo( x, y + 2.f*diameter);
        wings.lineTo( x + 2.f*diameter , y + 2.5f*diameter);
        wings.lineTo( x, y + diameter);
        wings.closePath();
        g2.setPaint(status);
        g2.fill(at.createTransformedShape(wings));
        g2.draw(at.createTransformedShape(wings));
        g2.setPaint(Color.black);
    }
}
```

D. GUI

```
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import java.util.*;
import java.awt.event.*;
/**
 * Title: Agent Economy
 * Description: Simulation of Searching Robots
 * Copyright:    Copyright (c) 2003
 * Company: NPS
 * @author: Monty Williams Original GUI code written by Asim Tokgoz adapted
 * for the use in this simulation.
 * @version 1.0
```

```
 */
public class GUI extends JPanel implements Data {
    Environment demo;
    JTabbedPane tabs = new JTabbedPane();
    // two main panels for tabbed panes
    JPanel firstPanel = new JPanel();
    JPanel secondPanel = new JPanel();
    // Panels to arrange buttons for tabbed pane 1
    JPanel DataPanel = new JPanel();
    //Buttons on the first tabbed pane
    JLabel numberOfXRobotsL = new JLabel("Number of Robots");
    JLabel numberOfX = new JLabel();
    JLabel numberOfYRobotsL = new JLabel("Number of y Robots");
    JLabel numberOfY = new JLabel();
    JLabel numberOfZRobotsL = new JLabel("Number of z Robots");
    JLabel numberOfZ = new JLabel();
    JButton startSimulation = new JButton("New Simulation");
   JToggleButton stopSimulation = new JToggleButton("Stop Simulation");
    // Sliders for tabbed pane 2
    JSlider xRobots ;
    JSlider yRobots ;
    JSlider zRobots ;
    JCheckBox swarmingCheck;
    JCheckBox followCheck;
```

```
  JCheckBox searchCheck;
/**
 * constructor for GUI
 */
public GUI() {
  try {
    jbInit();
  }
  catch(Exception e) {
    e.printStackTrace();
  }
}
/**
 * initialize GUI
 * @throws Exception
 */
private void jbInit() throws Exception {
  demo = new Environment();
  //init graph
 demo.setSize(550, 500);
  // tabbed panes
  tabs.add("Environment", firstPanel);
  tabs.add("Adjust parameters", secondPanel);
  this.setLayout(new BorderLayout());
```

```
this.add(tabs,BorderLayout.CENTER);
// first tabbed pane and elements of it
firstPanel.setLayout(null);
Insets insetsFirstPanel = firstPanel.getInsets();
firstPanel.add(demo);
firstPanel.add(DataPanel);
demo.setBounds(5+insetsFirstPanel.left,5 + insetsFirstPanel.top, 505, 505);
DataPanel.setBounds(535+insetsFirstPanel.left,5 + insetsFirstPanel.top, 250,550);
DataPanel.setSize(250, 550);
DataPanel.setLayout(null);
Insets insets = DataPanel.getInsets();
DataPanel.add(numberOfXRobotsL);
DataPanel.add(numberOfX);
numberOfXRobotsL.setBounds(20+insets.left,1 + insets.top, 150,25);
numberOfX.setBounds(155+insets.left,1 + insets.top, 40,25);
xRobots = new JSlider(JSlider.HORIZONTAL, 0,100, Data.numberOfXRobots);
xRobots.setMajorTickSpacing(20);
xRobots.setMinorTickSpacing(5);
xRobots.setPaintTicks(true);
xRobots.setPaintLabels(true);
DataPanel.add(xRobots);
xRobots.setBounds(10+insets.left,30 + insets.top, 200,25);
numberOfX.setText(String.valueOf( Data.numberOfXRobots ));
numberOfYRobotsL.setBounds(20+insets.left,75 + insets.top, 150,25);
```

```
numberOfY.setBounds(155+insets.left,75 + insets.top, 40,25);
yRobots = new JSlider(JSlider.HORIZONTAL, 0,100, Data.numberOfYRobots);
yRobots.setMajorTickSpacing(20);
yRobots.setMinorTickSpacing(5);
yRobots.setPaintTicks(true);
yRobots.setPaintLabels(true);
yRobots.setBounds(10+insets.left,105 + insets.top, 200,25);
numberOfY.setText(String.valueOf( Data.numberOfYRobots ));
numberOfZRobotsL.setBounds(20+insets.left,150 + insets.top, 150,25);
numberOfZ.setBounds(155+insets.left,150 + insets.top, 40,25);
zRobots = new JSlider(JSlider.HORIZONTAL, 0,100, Data.numberOfZRobots);
zRobots.setMajorTickSpacing(20);
zRobots.setMinorTickSpacing(5);
zRobots.setPaintTicks(true);
zRobots.setPaintLabels(true);
zRobots.setBounds(10+insets.left,180 + insets.top, 200,25);
numberOfZ.setText(String.valueOf( Data.numberOfZRobots ));
swarmingCheck = new JCheckBox("Swarming");
DataPanel.add(swarmingCheck);
swarmingCheck.setBounds(20+insets.left, 225 + insets.top, 100,25);
followCheck = new JCheckBox("Follow");
DataPanel.add(followCheck);
followCheck.setBounds(20+insets.left, 250 + insets.top, 100,25);
searchCheck = new JCheckBox("Search");
```

```
DataPanel.add(searchCheck);
searchCheck.setBounds(20+insets.left, 275 + insets.top, 100,25);
DataPanel.add(startSimulation);
startSimulation.setBounds(20+insets.left, 325 + insets.top, 130,30);
DataPanel.add(stopSimulation);
stopSimulation.setBounds(20+insets.left, 370 + insets.top, 130,30);
//action listeners
startSimulation.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e) {
    startSimulation_actionPerformed(e);
  }
});
stopSimulation.addActionListener(new java.awt.event.ActionListener() {
  public void actionPerformed(ActionEvent e) {
    stopSimulation_actionPerformed(e);
  }
});
xRobots.addChangeListener( new ChangeListener()
{
    public void stateChanged(ChangeEvent ce)
    {
        Data.numberOfXRobots = xRobots.getValue();
        numberOfX.setText(String.valueOf( Data.numberOfXRobots ));
```

```
    }
});
yRobots.addChangeListener( new ChangeListener()
{
    public void stateChanged(ChangeEvent ce)
    {
        Data.numberOfYRobots = yRobots.getValue();
        numberOfY.setText(String.valueOf( Data.numberOfYRobots ));

    }
});
zRobots.addChangeListener( new ChangeListener()
{
    public void stateChanged(ChangeEvent ce)
    {
        Data.numberOfZRobots = zRobots.getValue();
        numberOfZ.setText(String.valueOf( Data.numberOfZRobots ));

    }
});
swarmingCheck.addChangeListener( new ChangeListener()
{
    public void stateChanged(ChangeEvent ce)
    {
```

```
        if(swarmingCheck.isSelected()){
            Data.swarm = true;
        }
        else{
            Data.swarm = false;
        }
    }
});
followCheck.addChangeListener( new ChangeListener()
{
    public void stateChanged(ChangeEvent ce)
    {
        if(followCheck.isSelected()){
            Data.follow = true;
        }
        else{
            Data.follow = false;
        }
    }
});
searchCheck.addChangeListener( new ChangeListener()
{
    public void stateChanged(ChangeEvent ce)
    {
```

```
        if(searchCheck.isSelected()){
            Data.search = true;
        }
        else{
            Data.search = false;
        }
    }
  });
}
/**
 * This method arranges the sliders on the tabbed pane 2.
 */
public void sliders(){
  secondPanel.setLayout(null);
  /*consultant = new JCheckBox("Consultant");
  consultant.setBounds(480,190,100,40);
  secondPanel.add(consultant);*/
}
/**
 * Actions of start simulation button
 */
void startSimulation_actionPerformed(ActionEvent e) {
  demo.init();
  demo.start();
```

```
  }
  /**
   * Actions of agen selection button
   */
  void stopSimulation_actionPerformed(ActionEvent e) {
     if(stopSimulation.isSelected())
       demo.stop();
    else
       demo.start();
  }
}
```

E. DATA

```
/**
 * Title: Agent Economy
 * Description: Simulation of Searching Robots
 * Copyright:    Copyright (c) 2003
 * Company: NPS
 * @author: Monty Williams
 * @version 1.0
 */

interface Data {
  //simulation variables
  class DataClass {
    static int numberOfXRobots = 3;
```

```
    static int numberOfYRobots = 0 ;
    static int numberOfZRobots  = 0;
    static boolean swarm = false;
    static boolean follow = false ;
    static boolean search = false ;
    static short courses [] = { 0,45,90,135,180,225,270,315 };
    static int gridSize = 20;
    static int numberOfGrids = 500/gridSize;
    static short collision [][] = new short [numberOfGrids]  [numberOfGrids ];
  }
  DataClass Data = new DataClass();
}
```

F. SIMULATION RAW DATA

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	120	36	1	0.2	2	1	-37.2
2	44	13.2	1	0.2	2	1	-14.4
3	78	23.4	1	0.2	2	1	-24.6
4	45	13.5	1	0.2	2	1	-14.7
5	39	11.7	1	0.2	2	1	-12.9
6	55	16.5	1	0.2	2	1	-17.7
7	59	17.7	1	0.2	2	1	-18.9
8	120	36	1	0.2	2	1	-37.2
9	51	15.3	1	0.2	2	1	-16.5
10	79	23.7	1	0.2	2	1	-24.9
							-21.9

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	120	36	2	0.4	2	4	-40.4
2	43	12.9	2	0.4	2	4	-17.3
3	91	27.3	2	0.4	2	4	-31.7
4	39	11.7	2	0.4	2	4	-16.1
5	120	36	2	0.4	2	4	-40.4
6	120	36	2	0.4	2	4	-40.4
7	120	36	2	0.4	2	4	-40.4
8	48	14.4	2	0.4	2	4	-18.8
9	41	12.3	2	0.4	2	4	-16.7
10	70	21	2	0.4	2	4	-25.4
							-28.76

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	31	9.3	3	0.6	2	0	-9.9
2	23	6.9	3	0.6	2	0	-7.5
3	54	16.2	3	0.6	2	0	-16.8
4	35	10.5	3	0.6	2	0	-11.1
5	105	31.5	3	0.6	2	0	-32.1
6	29	8.7	3	0.6	2	0	-9.3
7	33	9.9	3	0.6	2	0	-10.5
8	21	6.3	3	0.6	2	0	-6.9
9	19	5.7	3	0.6	2	0	-6.3
10	21	6.3	3	0.6	2	0	-6.9
							-11.73

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	35	10.5	4	0.8	2	0	-11.3
2	23	6.9	4	0.8	2	0	-7.7
3	24	7.2	4	0.8	2	0	-8
4	45	13.5	4	0.8	2	0	-14.3
5	39	11.7	4	0.8	2	0	-12.5
6	29	8.7	4	0.8	2	0	-9.5
7	38	11.4	4	0.8	2	0	-12.2
8	28	8.4	4	0.8	2	0	-9.2
9	26	7.8	4	0.8	2	0	-8.6
10	87	26.1	4	0.8	2	0	-26.9
							-12.02

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	120	36	5	1	2	3	-40
2	44	13.2	5	1	2	3	-17.2
3	78	23.4	5	1	2	3	-27.4
4	45	13.5	5	1	2	3	-17.5
5	39	11.7	5	1	2	3	-15.7
6	55	16.5	5	1	2	3	-20.5
7	59	17.7	5	1	2	3	-21.7
8	120	36	5	1	2	3	-40
9	51	15.3	5	1	2	3	-19.3
10	79	23.7	5	1	2	3	-27.7
							-24.7

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	120	36	6	1.2	2	5	-42.2
2	120	36	6	1.2	2	5	-42.2
3	120	36	6	1.2	2	5	-42.2
4	120	36	6	1.2	2	5	-42.2
5	120	36	6	1.2	2	5	-42.2
6	120	36	6	1.2	2	5	-42.2
7	120	36	6	1.2	2	5	-42.2
8	120	36	6	1.2	2	5	-42.2
9	120	36	6	1.2	2	5	-42.2
10	120	36	6	1.2	2	5	-42.2
							-42.2

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	120	36	7	1.4	2	4	-41.4
2	120	36	7	1.4	2	5	-42.4
3	120	36	7	1.4	2	5	-42.4
4	120	36	7	1.4	2	5	-42.4
5	120	36	7	1.4	2	5	-42.4
6	120	36	7	1.4	2	5	-42.4
7	120	36	7	1.4	2	5	-42.4
8	120	36	7	1.4	2	5	-42.4
9	120	36	7	1.4	2	5	-42.4
10	120	36	7	1.4	2	5	-42.4
							-42.3

run	alpha	c1alpha	beta	c2beta	zeta	c3zeta	fom
1	120	36	8	1.6	2	6	-43.6
2	120	36	8	1.6	2	6	-43.6
3	120	36	8	1.6	2	6	-43.6
4	120	36	8	1.6	2	6	-43.6
5	120	36	8	1.6	2	6	-43.6
6	120	36	8	1.6	2	6	-43.6
7	120	36	8	1.6	2	6	-43.6
8	120	36	8	1.6	2	6	-43.6
9	120	36	8	1.6	2	6	-43.6
10	120	36	8	1.6	2	6	-43.6
							-43.6

THIS PAGE INTENTIONALLY BLANK

APPENDIX B. BENDER CONTROL CODE

In this appendix the controlling program for Bender is submitted. The program is written in Java using JBuilder 5 Personal edition. The program consists of the following classes:

- Bender
- BenderGUI
- Compass
- GPS
- ManualGUI
- Motors and
- Sensors

A. BENDER

```
/**

 * <p>Title: SE4015 Final Project</p>

 * <p>Description: Final implementation of Bender using Brooks Subsumption architecture for collision avoidance and waypoint navigation.</p>

 * <p>Copyright: Copyright (c) 2002</p>

 * <p>Company: NPS</p>

 * @author Monty Williams

 * @version 1.0

 */



 import java.io.IOException;
```

```
public class Bender extends Thread
{
  private final float DISTANCE_THRESHOLD = 3.0f;
  private final float COURSE_THRESHOLD = 6.3f;
  //******** MOTOR CONTROLLER ********//
  private Motors motor;
  //******** SENSOR SUITE ********//
  private GPS gps;
  private Compass compass;
  private Sensors sensor;
  //******** STATE VARIABLES ********//
  private boolean done;
  private boolean runBot;
  //Tokens used to pass control to different control methods
  private boolean wpToken;
  private boolean avoidToken;
  //Compass heading of the robot
  private float heading;
```

```
private float desiredHeading;

private float diffInHeading;

//Internal state of the motors

// 0 = stop, 1 = forward, 2 =  turn

private int motorState;

public Bender()

{

  //Instantiate the motor getMotor() returns

  //the singleton

  motor = Motors.getMotor();

  compass = new Compass();

  gps = new GPS();

  sensor = new Sensors();

  heading = compass.getHeading();

}//end constructor

/**

 *

 */

 public void run()
```

```
{
  while(!done)
  {
    if(runBot)
    {
      //***** EXECUTING COMMANDS GO HERE *****//
      resolver();
      if(avoidToken)
      {
        avoidControl();
      }//end if
      else if (wpToken)
      {
        wpControl();
      }//end else-if
      else
      {
        cruiseControl();
      }//end else
```

```
    }//end if
   }//end while loop
}//end method run
/**
 *
 */
 public void stop(boolean stop)
 {
    if(stop == true)
    {
      done = true;
    }//end if
 }//end method stop
 /**
  *
  */
  public void runBender()
  {
    runBot = true;
```

```
}//end method runBender
/**
 *
 */
 public void stopBender()
 {
  try
  {
    motor.move(0);
    motorState = 0;
    runBot = false;
  }
  catch (IOException ie)
  {}
 }//end method stopBender
/**
 *
 */
 private void resolver()
```

```
            {
             gps.queryGPSReceiver();
             //query the sensors
             sensor.getSensors();
             //Check for objects within the sensor range
             if(sensor.distance[0] < DISTANCE_THRESHOLD || sensor.distance[1] <
DISTANCE_THRESHOLD)
             {
              //set the avoid token to true
              avoidToken = true;
             }//end if
             //Get the current heading
             heading = compass.getHeading();
             desiredHeading = gps.calculateHeading();
             diffInHeading = Math.abs((heading - desiredHeading));
             System.out.println("The difference is " + diffInHeading);
             //if(heading > 361.0f)
               //heading = 0.0f;
               /*
```

```
        //Check the difference between desired and actual course
        if(diffInHeading > COURSE_THRESHOLD)
        {
          //Set the wpToken to trye
          wpToken = true;
        }//end if
*/
      }//end method resolver
      /**
       *
       */
      private void cruiseControl()
      {
        if(motorState != 1)
        {
          try
          {
            motor.move(1);
            motorState = 1;
```

```
      BenderGUI.logReport("Issued forward command");
    }
    catch (IOException ie)
    {}
  }//end if
}//end method cruiseControl
/**
 *
 */
 private void wpControl()
 {
  BenderGUI.logReport("Inside waypoint control");
  //Stop the motors
  try
  {
   motor.move(0);
   motorState = 0;
  }
  catch (IOException ie)
```

```
{}
//Calculate the heading needed to get to waypoint
desiredHeading = gps.calculateHeading();
BenderGUI.logReport(Float.toString(desiredHeading));
compass.sendDesiredHeading(desiredHeading);
motorState = 2;
//Get the current heading
heading = compass.getHeading();
float difference = Math.abs( (desiredHeading - heading) );
/*
//******** LET JAVA HANDLE THE TURN ********
//Compare the directions of the heading
if(desiredHeading > heading)
{
  //Compare difference to 180 degrees to ensure
  //that Bender turns in the right direction
  if( difference < 180 )
  {
    //Issue a turn right command
```

```
                try
                {
                  motor.move(3);
                  motorState = 2;
                  BenderGUI.logReport("Issued a right turn command inside
wpControl()");
                }
                catch ( IOException ie)
                {}
              }//end if
              //Difference between the to headings is greater than 180 degrees
              //so turn in the opposite direction
              else
              {
                //Issue a turn left command
                try
                {
                  motor.move(2);
                  motorState = 2;
```

```
                BenderGUI.logReport("Issued a left turn command inside wpControl()");
                }
                catch (IOException ie)
                {}
              }//end else
            }//end if
            //The desired heading was less than the current heading
            //so we'll do just the opposite of the above code
            else
            {
              //If the difference is less than 180 degrees
              //then turn Bender in the left direction
              if( difference < 180)
              {
                //Issue left turn command
                try
                {
                  motor.move(2);
```

```
                motorState = 2;
                BenderGUI.logReport("Issued a left turn command inside wpControl()");
              }
              catch (IOException ie)
              {}
            }//end if
            //The difference was greater than 180 degrees
            //so turn Bender in the right direction
            else
            {
              //Issue right turn command
              try
              {
                motor.move(2);
                motorState = 2;
                BenderGUI.logReport("Issued a right turn command inside wpControl()");
              }
              catch (IOException ie)
```

```
    {}

   }//end else


  }//end else


  //******** END OF JAVA TURN TO WAYPOINT ********/

  //Loop through while the difference

  while(difference > COURSE_THRESHOLD)

  {

   //Get the absolute difference between the current heading and

   //the desired heading

   difference = Math.abs( (compass.getHeading() - desiredHeading) );

   BenderGUI.logReport((“Turning to course inside waypoint : “ +
Float.toString(desiredHeading)));

  }//end while loop

  //Issue a stop command

  try

  {

   motor.move(0);
```

```
    motorState = 0;
    BenderGUI.logReport("Issued a stop turn command inside wpControl()");
   }
   catch(IOException ie)
   {}
   //Set the waypoint token to false and exit
   wpToken = false;
          BenderGUI.logReport("Exiting waypoint control");
}//end method wpControl
/**
 *
 */
 private void avoidControl()
 {
    //Stop issue stop command
    try
    {
      motor.move(0);
      motorState = 0;
```

```
                BenderGUI.logReport("Issued STOP command inside avoidControl()");
            }
            catch (IOException ie)
            {}
            sensor.getSensors();
            while(avoidToken)
            {
              if(motorState != 2)
              {
                if((sensor.distance[1] < sensor.distance[0]) || sensor.distance[4] < sensor.distance[5])
                {
                  try
                  {
                    motor.move(2);
                    BenderGUI.logReport("Issued a left turn inside avoidControl()");
                  }
                  catch (IOException ie)
                  {}
```

```
                    }//end if
                    else
                    {
                     try
                     {
                       motor.move(3);
                       BenderGUI.logReport("Issued a right turn inside avoidControl()");
                     }
                     catch (IOException ie)
                     {}
                    }//end else
                    motorState = 2;
                  }//end if
                  sensor.getSensors();
                  if(sensor.distance[0] > DISTANCE_THRESHOLD || sensor.distance[1]
> DISTANCE_THRESHOLD)
                  {
                   avoidToken = false;
                  }
```

```
        }//end while

      }//end method avoidControl

}//end clas Bender
```

B. BENDERGUI

```
/**

 * <p>Title: SE4015 Final Project</p>

 * <p>Description: Final implementation of Bender using Brooks Subsumption architecture for collision avoidance and waypoint navigation.</p>

 * <p>Copyright: Copyright (c) 2002</p>

 * <p>Company: NPS</p>

 * @author Monty Williams

 * @version 1.0

 */

 import java.awt.*;

 import java.awt.event.*;

 import javax.swing.*;

public class BenderGUI extends JFrame

{

  //******** SWING COMPONENTS FOR THE LAYOUT OF THE APPLICATION ********//
```

```
//Text area for messages to the user
private static JTextArea console;
private JLabel consoleLabel;
//Buttons to start and stop the robot
private JButton runButton;
private JButton stopButton;
//Waypoint GUI components
private JLabel wpLabel;
//Labels for the lattitude degrees and minutes
private JLabel latDegLabel;
private JLabel latMinLabel;
//Labels for the longitude degrees and minutes
private JLabel lonDegLabel;
private JLabel lonMinLabel;

//Text fields for the lattitude degrees and minutes
private JTextField latDegField;
private JTextField latMinField;
//Text fields for the longitude degrees and minutes
```

```
private JTextField lonDegField;

private JTextField lonMinField;

//Check box to commence navigation to waypoint

private JCheckBox navCheckBox;

private ManualGUI manual;

//******** Bender Components ********//

private Bender bender;

public BenderGUI()

{

 Container container = getContentPane();

 JTabbedPane tabs = new JTabbedPane();

 //manual = new ManualGUI();

 container.add(tabs, BorderLayout.CENTER);

 //Get the content pane to add components to

 JPanel cp = new JPanel(new BorderLayout());

 tabs.addTab("Autonomous Control", cp);

 //tabs.addTab("Manual Control", manual);

 //Box component used to hold the waypoint gui

 Box rightBox = Box.createVerticalBox();
```

```
//Instantiate the waypoint label and add it to the box
//Struts are used to create vertical space between components
wpLabel = new JLabel("Waypoint Data");
rightBox.add(Box.createVerticalStrut(15));
rightBox.add(wpLabel);
rightBox.add(Box.createVerticalStrut(20));
//Instantiate the label and field for lat degrees and
//add them to the box
latDegLabel = new JLabel("Lattitude Degrees");
latDegField = new JTextField(10);
rightBox.add(latDegLabel);
rightBox.add(latDegField);
rightBox.add(Box.createVerticalStrut(10));
//Instantiate the label and field for the lat minutes
//and add them to the box
latMinLabel = new JLabel("Lattitude Minutes");
latMinField = new JTextField(10);
rightBox.add(latMinLabel);
rightBox.add(latMinField);
```

```
rightBox.add(Box.createVerticalStrut(10));

//Instantiate the label and field for the lon degrees

//and add them to the box

lonDegLabel = new JLabel("Longitude Degrees");

lonDegField = new JTextField(10);

rightBox.add(lonDegLabel);

rightBox.add(lonDegField);

rightBox.add(Box.createVerticalStrut(10));

//Instantiate the label and field for the lon minutes

//and add them to the box

lonMinLabel = new JLabel("Longitude Minutes");

lonMinField = new JTextField(10);

rightBox.add(lonMinLabel);

rightBox.add(lonMinField);

rightBox.add(Box.createVerticalStrut(10));

//Instantiate the check box and add them to the box

navCheckBox = new JCheckBox("Navigate to Waypoint", false);

rightBox.add(navCheckBox);

//Add the box to the west area of the content pane
```

```
cp.add(rightBox, BorderLayout.EAST);

//Instantiate a JPanel to add components to

JPanel southBox = new JPanel(new FlowLayout());

//Instantiate the control buttons for the application

runButton = new JButton(“Ron”);

stopButton = new JButton(“Stop”);

//Add the buttons to the JPanel

southBox.add(runButton);

southBox.add(Box.createHorizontalStrut(20));

southBox.add(stopButton);

//create a vertical box for the center panel

Box centerBox = Box.createVerticalBox();

//Instantiate the console label and the console

//add them to the box

consoleLabel = new JLabel(“Console”);

console = new JTextArea(“>>“,15,30);

centerBox.add(consoleLabel);

centerBox.add(new JScrollPane(console));

//add the JPanel to the center box
```

```
centerBox.add(southBox);


//add the center box to the center area of the content pane
cp.add(centerBox, BorderLayout.CENTER);
manual = new ManualGUI();
tabs.addTab("Manual Control", manual);
//Set the size of the JFrame
setSize(650, 300);
//Make the JFrame visible
setVisible(true);
//Add a window closing event and handle any closing actions as required
addWindowListener(new WindowAdapter()
{
 public void windowClosing( WindowEvent event )
 {
     //***** ADD WINDOW CLOSING EVENTS HERE *****//
  bender.stop(true);
     System.exit(0);
 }
```

```
});//end addWindowListener method



//***** Handle the run button events *****//
runButton.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent ae)
  {
       //***** ADD BUTTON EVENTS HERE *****//
    bender.runBender();
  }
});//end of runButton action listener
//***** Handle the stop button events *****//
stopButton.addActionListener(new ActionListener()
{
  public void actionPerformed(ActionEvent ae)
  {
       //***** ADD BUTTON EVENTS HERE *****//
    bender.stopBender();
```

```
    }
  });//end of stopButton action listener
  bender = new Bender();
  bender.start();
}//end constructor


/**
 *
 */
 public static void logReport(String report)
 {
    //Print the report to the console window
    console.append(report + "\n >>");
 }//end method logReport
/**
 *
 */
 public static void main(String [] args)
 {
```

```
        BenderGUI application = new BenderGUI();

        application.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

      }//end main

    }//end class BenderGUI
```

C. COMPASS

```
    /**

     * <p>Title: SE4015 Final Project</p>

     * <p>Description: Final implementation of Bender using Brooks Subsumption architecture for collision avoidance and waypoint navigation.</p>

     * <p>Copyright: Copyright (c) 2002</p>

     * <p>Company: NPS</p>

     * @author Monty Williams

     * @version 1.0

     */

     import java.io.*;

     import java.net.*;

    public class Compass

    {

      //******** Connection parameters ********//
```

```
private static final String IP = “131.120.101.81”;

private static final int COMPASS_PORT = 2001;

//******** Network interface variables ********//

private Socket compassSocket;

private BufferedReader input;

private OutputStream output;


//*** String processing variable ***//

private String rawCompassData;

private float course;

public Compass()

{

 try

 {

  //Instantiate the socket

  compassSocket = new Socket(IP, COMPASS_PORT);

  //Instantiate the IO streams from the socket

  input = new BufferedReader(new
InputStreamReader(compassSocket.getInputStream()));
```

```
    output = compassSocket.getOutputStream();
    //Diagnostic to let user know that the socket was connected
    BenderGUI.logReport("Compass : Connected");
   }
   catch (IOException ie)
   {
    //Log the error in the console
    BenderGUI.logReport(("Compass : " + ie));
   }
  }//end constructor
  /**
   *
   */
   public float getHeading()
   {
    try
    {
     rawCompassData = input.readLine();
    }
```

```
  catch (IOException ie)
  {}
  BenderGUI.logReport(rawCompassData);
  if(rawCompassData.length() > 1)
  {
    //return Float.parseFloat(rawCompassData);
    try
    {
      course = Float.parseFloat(rawCompassData) - 10.0f;
    }
    catch (NumberFormatException e)
    {}
    if(course < 0)
     course += 360.0f;
    return course;
  }
  else
    return course;
}//end method getHeading
```

```
/**
 *
 */
 public void sendDesiredHeading(float dHead)
 {
    String head = Float.toString(dHead);
    try
    {
      output.write((head + "\n").getBytes());
    }
    catch (IOException ie)
    {}
 }//end method sendDesiredHeading
 /**
  *
  */
  public void close()
  {
    try
```

```
            {
             input.close();
             output.close();
             compassSocket.close();
            }
           catch (IOException ie)
           {}
         }//end method close
      }//end class Compass
```

D. GPS

```
      /**
       * <p>Title: SE4015 Final Project</p>
       * <p>Description: Final implementation of Bender using Brooks Subsumption architecture for collision avoidance and waypoint navigation.</p>
       * <p>Copyright: Copyright (c) 2002</p>
       * <p>Company: NPS</p>
       * @author Monty Williams
       * @version 1.0
       */
```

```
import java.io.*;

import java.net.*;

import java.util.StringTokenizer;

public class GPS

{

 public static final String IP = “131.120.101.81”;

 public static final int GPS_PORT = 2002;

 public static final String DELIMITTER = “,”;

 //***** NETWORK INSTANCE VARIABLES TO CONNECT TO THE ROBOT *****//

 private Socket socket;

 private BufferedReader gpsSockIn;

 //***** GPS LATTITUDE AND LONGITUDE VARIABLES TO PASS BACK TO CALLING SYSTEM  *****//

 private float latDegrees;

 private float latMinutes;

 private float lonDegrees;

 private float lonMinutes;

 private float wpLatDegrees;

 private float wpLatMinutes;
```

```
private float wpLonDegrees;

private float wpLonMinutes;

private float latMeters;

private float lonMeters;

private float wpLatMeters;

private float wpLonMeters;

private float latDiff;

private float lonDiff;

private String tempString;

private String rawGPSString;

private float courseToSteer;

public GPS()

{

  wpLatDegrees = 36.0f;

  wpLatMinutes = 35.716f;

  wpLonDegrees = 121.0f;

  wpLonMinutes = 52.5002f;

  //Connect to the GPS port

  try
```

```
            {
              socket = new Socket(IP, GPS_PORT);
              gpsSockIn = new BufferedReader( new InputStreamReader(
socket.getInputStream() ) );
              BenderGUI.logReport("GPS: Connected");
            }
            catch (IOException ie)
            {
              //Log the error in the console
              BenderGUI.logReport(("GPS : " + ie));
            }//end try/catch block
          }//end constructor
          /**
           *
           */
          public void queryGPSReceiver()
          {
            try
            {
```

```
    //Extract new GPS data
    rawGPSString = gpsSockIn.readLine();
    BenderGUI.logReport("GPS Data : " + rawGPSString);
   }
   catch(IOException ie)
   {}
   System.out.println("GPS Data : " + rawGPSString);
   if(rawGPSString.length() > 1)
   {
    StringTokenizer st = new StringTokenizer(rawGPSString, DELIMITTER);
    latDegrees = Float.parseFloat(st.nextToken());
    latMinutes = Float.parseFloat(st.nextToken());
    tempString = st.nextToken();
    lonDegrees = Float.parseFloat(st.nextToken());
    lonMinutes = Float.parseFloat(st.nextToken());
   }//end if
  }//end method queryGPSReceiver()
  /**
   *
```

```
*/
public void setWaypoint(float latDeg, float latMin, float lonDeg, float lonMin)
{
  wpLatDegrees = latDeg;
  wpLatMinutes = latMin;
  wpLonDegrees = lonDeg;
  wpLonDegrees = lonMin;
}//end method setWaypoint
/**
 *
 */
 public float calculateHeading()
 {
  queryGPSReceiver();
  latMeters = (float)(latDegrees * 60 + latMinutes) * 1852;
  lonMeters = (float)-(lonDegrees * 60 + lonMinutes) * 1852 *
(float)Math.cos(.628);
  wpLatMeters = (wpLatDegrees * 60 + wpLatMinutes) * 1852;
  wpLonMeters = -(wpLonDegrees * 60 + wpLonMinutes) * 1852 *
(float)Math.cos(.628);
```

```
latDiff = wpLatMeters - latMeters;
lonDiff = wpLonMeters - lonMeters;
if(lonDiff == 0)
  lonDiff = 0.001f;
if(latDiff == 0)
  latDiff = 0.001f;
if(lonDiff >= 0 && latDiff >= 0)
   courseToSteer = (float)(180 / Math.PI * Math.atan(lonDiff / latDiff));
else if(lonDiff >= 0 && latDiff <= 0)
   courseToSteer = (float)(90 - 180 /Math.PI * Math.atan(latDiff/lonDiff));
else if(lonDiff <= 0 && latDiff >= 0)
   courseToSteer =    (float)(360 + 180 / Math.PI * Math.atan(lonDiff /
latDiff));
else
   courseToSteer = (float)(180 + 180 /Math.PI * Math.atan(latDiff / lonDiff));
BenderGUI.logReport(Float.toString(courseToSteer));
//courseToSteer -= 90.0f;
//if(courseToSteer < 0)
//courseToSteer += 360.0f;
```

```
        return (courseToSteer);
      }//end method calculateHeading
      /**
     *
     */
     public void close()
     {
       try
       {
         gpsSockIn.close();
         socket.close();
       }
       catch (IOException ie)
       {}
     }//end method close
}//end class GPSThread
```

E. MANUALGUI

```
/**
 * <p>Title: SE4015 Final Project</p>
```

```
 * <p>Description: Final implementation of Bender using Brooks Subsumption architecture for collision avoidance and waypoint navigation.</p>

 * <p>Copyright: Copyright (c) 2002</p>

 * <p>Company: NPS</p>

 * @author Monty Williams

 * @version 1.0

 */

 import java.awt.*;

 import java.awt.event.*;

 import javax.swing.*;

 import javax.swing.event.*;

 import java.io.IOException;

 import java.util.Hashtable;

public class ManualGUI extends JPanel

{

  public static final int REVERSE = 100;

  public static final int STOP    = 150;

  public static final int FORWARD = 200;

  public static final int RANGE   = FORWARD - REVERSE;
```

```
//******** MOTOR CONTROLLER ********//
private Motors motor;
//******** GUI COMPONENTS ********//
private JPanel controlPanel;
private JPanel buttonPanel;
private JSlider leftMotorSlider;
private JSlider rightMotorSlider;
private JSlider bothMotorSlider;
private JLabel sliderLabel;
private JButton forwardButton;
private JButton reverseButton;
private JButton stopButton;
private JButton leftButton;
private JButton rightButton;
private GridBagLayout gbl;
private GridBagConstraints gbc;
private Hashtable labelTable;
public ManualGUI()
{
```

```
setLayout(new BorderLayout());

 motor = Motors.getMotor();

gbl = new GridBagLayout();

gbc = new GridBagConstraints();

controlPanel = new JPanel(gbl);

buttonPanel  = new JPanel(gbl);

labelTable = new Hashtable();

labelTable.put(new Integer(REVERSE), new JLabel("R"));

labelTable.put(new Integer(STOP), new JLabel("S"));

labelTable.put(new Integer(FORWARD), new JLabel("F"));

//Instantiate the left motor slider

leftMotorSlider = new JSlider(JSlider.HORIZONTAL, REVERSE,
FORWARD, STOP);



//Set the attributes of the slider

leftMotorSlider.setMajorTickSpacing(50);

leftMotorSlider.setMinorTickSpacing(5);

leftMotorSlider.setSnapToTicks(true);

leftMotorSlider.setPaintTicks(true);
```

```
leftMotorSlider.setLabelTable(labelTable);

leftMotorSlider.setPaintLabels(true);

leftMotorSlider.setBorder(BorderFactory.createEmptyBorder(0, 0, 10, 0));

//Set the constraints for adding the slider to the panel

gbc.gridx       = 0;

gbc.gridwidth   = 2;

gbc.gridy       = 0;

gbc.gridheight  = 2;

sliderLabel = new JLabel("Left Motor");

gbl.setConstraints(sliderLabel, gbc);

controlPanel.add(sliderLabel);

gbc.gridx  = 2;

gbc.gridy  = 0;

gbc.setConstraints(leftMotorSlider, gbc);

controlPanel.add(leftMotorSlider);


leftMotorSlider.addChangeListener(new ChangeListener(){

  public void stateChanged(ChangeEvent ce){

    int val = leftMotorSlider.getValue();
```

```
            //***** ADD MOTOR COMMAND HERE *****//
            try
                {
                motor.move(2);
                }
                catch(IOException io)
                {}
             }
        });
        //Instantiate the right motor slider
        rightMotorSlider = new JSlider(JSlider.HORIZONTAL, REVERSE,
FORWARD, STOP);
        //Set the attributes of the slider
        rightMotorSlider.setMajorTickSpacing(50);
        rightMotorSlider.setMinorTickSpacing(5);
        rightMotorSlider.setSnapToTicks(true);
        rightMotorSlider.setPaintTicks(true);
        rightMotorSlider.setLabelTable(labelTable);
        rightMotorSlider.setPaintLabels(true);
```

```
rightMotorSlider.setBorder(BorderFactory.createEmptyBorder(0, 0, 10, 0));
//Set the constraints for adding the slider to the panel
gbc.gridx       = 0;
gbc.gridwidth   = 2;
gbc.gridy       = 2;
gbc.gridheight  = 2;
sliderLabel = new JLabel("Right Motor");
gbl.setConstraints(sliderLabel, gbc);
controlPanel.add(sliderLabel);
gbc.gridx  = 2;
gbc.gridy  = 2;
gbl.setConstraints(rightMotorSlider, gbc);
controlPanel.add(rightMotorSlider);
rightMotorSlider.addChangeListener(new ChangeListener(){
  public void stateChanged(ChangeEvent ce){
    int val = rightMotorSlider.getValue();
    //***** ADD MOTOR COMMAND HERE *****//
    try
          {
```

```
                motor.move(3);
                }
                catch(IOException io)
                {}
            }
        });
        //Instantiate the right motor slider
        bothMotorSlider = new JSlider(JSlider.HORIZONTAL, REVERSE,
FORWARD, STOP);
        //Set the attributes of the slider
        bothMotorSlider.setMajorTickSpacing(50);
        bothMotorSlider.setMinorTickSpacing(5);
        bothMotorSlider.setSnapToTicks(true);
        bothMotorSlider.setPaintTicks(true);
        bothMotorSlider.setLabelTable(labelTable);
        bothMotorSlider.setPaintLabels(true);
        bothMotorSlider.setBorder(BorderFactory.createEmptyBorder(0, 0, 10, 0));
        //Set the constraints for adding the slider to the panel
        gbc.gridx      = 0;
```

```
gbc.gridwidth   = 2;
gbc.gridy       = 4;
gbc.gridheight  = 2;
sliderLabel = new JLabel("Both Motor");
gbl.setConstraints(sliderLabel, gbc);
controlPanel.add(sliderLabel);
gbc.gridx  = 2;
gbc.gridy  = 4;
gbl.setConstraints(bothMotorSlider, gbc);
controlPanel.add(bothMotorSlider);
bothMotorSlider.addChangeListener(new ChangeListener(){
  public void stateChanged(ChangeEvent ce){
    int val = bothMotorSlider.getValue();
    //***** ADD MOTOR COMMAND HERE *****//
  }
});
//Add the control panel to the center area
add(controlPanel, BorderLayout.WEST);
//******** CONTROL BUTTONS ********//
```

```
stopButton = new JButton("Stop");
  stopButton.addActionListener(
    new ActionListener() {
      public void actionPerformed(ActionEvent ev) {

        leftMotorSlider.setValue(STOP);
        rightMotorSlider.setValue(STOP);
        bothMotorSlider.setValue(STOP);
        try
        {
        motor.move(0);
        }
        catch(IOException io)
        {}
      }
    });
  gbc.gridx = 3;
  gbc.gridy = 3;
  gbc.gridwidth = 2;
```

```
gbc.gridheight = 2;

gbc.ipadx = 10;

gbc.ipady = 10;

gbc.insets = new Insets(10,0,0,0);

gbl.setConstraints(stopButton, gbc);

buttonPanel.add(stopButton);



//FORWARD BUTTON

forwardButton = new JButton("Forward");

forwardButton.addActionListener(

  new ActionListener() {

    public void actionPerformed(ActionEvent ev) {

      leftMotorSlider.setValue(FORWARD);

      rightMotorSlider.setValue(FORWARD);

      bothMotorSlider.setValue(FORWARD);

      try

      {

      motor.move(1);

      }
```

```
        catch(IOException io)
        {}
      }
    });
gbc.gridx = 3;
gbc.gridy = 1;
gbc.gridwidth = 2;
gbc.gridheight = 1;
gbc.ipadx = 20;
gbc.ipady = 20;
gbc.insets = new Insets(5,5,5,5);
gbl.setConstraints(forwardButton, gbc);
buttonPanel.add(forwardButton);
//LEFT TURN BUTTON
leftButton = new JButton("Left Turn");
leftButton.addActionListener(
  new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
      leftMotorSlider.setValue(STOP-50);
```

```
        rightMotorSlider.setValue(STOP+50);
        try
        {
        motor.move(2);
        }
        catch(IOException io)
        {}
      }
   });
gbc.gridx = 1;
gbc.gridy = 3;
gbc.gridheight = 4;
gbc.ipadx = 20;
gbc.ipady = 20;
gbl.setConstraints(leftButton, gbc);
buttonPanel.add(leftButton);
//RIGHT TURN BUTTON
rightButton = new JButton("Right Turn");
rightButton.addActionListener(
```

```
  new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
      leftMotorSlider.setValue(STOP+50);
      rightMotorSlider.setValue(STOP-50);
      try
      {
      motor.move(3);
      }
      catch(IOException io)
      {}
    }
  });
gbc.gridx = 5;
gbc.gridy = 3;
gbc.gridheight = 4;
gbc.ipadx = 20;
gbc.ipady = 20;
gbl.setConstraints(rightButton, gbc);
buttonPanel.add(rightButton);
```

```
//REVERSE BUTTON
    JButton reverseButton = new JButton("Reverse");
    reverseButton.addActionListener(
      new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          leftMotorSlider.setValue(REVERSE);
          rightMotorSlider.setValue(REVERSE);
          bothMotorSlider.setValue(REVERSE);
          try
          {
          motor.move(4);
          }
          catch(IOException io)
          {}
        }
      });
    gbc.gridx = 3;
    gbc.gridy = 7;
    gbc.gridwidth = 2;
```

```
        gbc.gridheight = 4;

        gbc.insets = new Insets(5,5,5,5);

        gbc.ipadx = 20;

        gbc.ipady = 20;

        gbl.setConstraints(reverseButton, gbc);

        buttonPanel.add(reverseButton);



        add(buttonPanel, BorderLayout.CENTER);

      }//end constructor

    }//end class ManualGUI
```

F. MOTORS

```
    /**

     * <p>Title: SE4015 Final Project</p>

     * <p>Description: Final implementation of Bender using Brooks Subsumption architecture for collision avoidance and waypoint navigation.</p>

     * <p>Copyright: Copyright (c) 2002</p>

     * <p>Company: NPS</p>

     * @author Monty Williams

     * @version 1.0
```

```
 */
 import java.io.*;
 import java.net.*;
public class Motors
{
  //***** NETWORK AND SOCKET INSANCE VARIABLES *****//
  public static final String IP = “131.120.101.81”;
  public static final int MOTOR_PORT = 2000;
  private static Socket socket;
  private static OutputStream motorCommandOut;
  private static Motors motor;
  //***** MOTOR CONTROL STATE VARIABLES *****//
  public static final int FORWARD = 175;
  public static final int STOP = 150;
  public static final int REVERSE = 75;
  /**
   *
   */
  private Motors()
```

```
{
 try
 {
  //Instantiate the socket
  socket = new Socket(IP, MOTOR_PORT);
  //Instantiate the output stream from the socket
  motorCommandOut = socket.getOutputStream();
  //Indicate to user that the motor socket connected
  BenderGUI.logReport("Motor: Connected");
 }
 catch(IOException ie)
 {
  //Log the error in the console
  BenderGUI.logReport(("Motors : " + ie));
 }//end try/catch block
}//end constructor
/**
 *
 */
```

```
public static Motors getMotor()
{
  if(motor == null)
  {
    motor = new Motors();
  }

  return motor;

}//end method getMotor()
/**
 *
 */
 public void stop()
 {
  try
  {
    motorCommandOut.close();
    socket.close();
```

```
    }
    catch(IOException ie)
    {
    }
  }//end method stop
/**
 *
 */
 public void close()
 {
    try
    {
      //Clean up by closing the
      //output stream and socket
      motorCommandOut.close();
      socket.close();
    }
    catch (IOException ie)
    {}
```

```
}//end method close
/**
 *
 */
 public static void move(int move) throws IOException
 {
   switch(move)
   {
        //Stop
        case 0:
            motorCommandOut.write(("x,3,\n").getBytes());
        break;

        //Forward
        case 1:
            motorCommandOut.write(("x,2," + FORWARD + ",\n").getBytes());
        break;

        //Left
```

```
        case 2:
            motorCommandOut.write((“x,0,” + REVERSE + “,\n”).getBytes());
            motorCommandOut.write((“x,1,” + FORWARD + “,\n”).getBytes());
        break;

        //Right
        case 3:
            motorCommandOut.write((“x,1,” + REVERSE + “,\n”).getBytes());
            motorCommandOut.write((“x,0,” + FORWARD + “,\n”).getBytes());
        break;
        //Reverse
        case 4:
            motorCommandOut.write((“x,2,” + REVERSE + “,\n”).getBytes());
        break;
        default:
            motorCommandOut.write((“x,3,” + REVERSE + “,\n”).getBytes());
    }//end switch
    motorCommandOut.flush();
}//end method manuever()
```

```
}//end class MotorThread
```

G. SENSORS

```
/**

 * <p>Title: SE4015 Final Project</p>

 * <p>Description: Final implementation of Bender using Brooks Subsumption architecture for collision avoidance and waypoint navigation.</p>

 * <p>Copyright: Copyright (c) 2002</p>

 * <p>Company: NPS</p>

 * @author Monty Williams

 * @version 1.0

 */

 import java.io.*;

 import java.net.*;

 import java.util.StringTokenizer;

public class Sensors

{//***** NETWORK AND SOCKET INSTANCE VARIABLES *****//

  public static final String IP = "131.120.101.81";

  public static final int SENSOR_PORT = 2003;

  private Socket socket;
```

```
private BufferedReader sensorSockIn;

private String sensorData;

public float [] distance;

public static final String DELIMITTER = ",";

public Sensors()

{

 //Declare an array of [6] floating point numbers

 //to hold the distances received by the sensor

 distance = new float[6];



 try

 {

  //Instantiate the socket

  socket = new Socket(IP, SENSOR_PORT);



  //Instantiate the input stream from the socket

  sensorSockIn = new BufferedReader(new
InputStreamReader(socket.getInputStream()));
```

```
  //Diagnostic to let user know that the socket was connected
  BenderGUI.logReport("Sensor : Connected");
 }

 catch(IOException ie)
 {
  //Log the error in the console
  BenderGUI.logReport(("Sensors : " + ie));

 }//end try/catch block

}//end constructor

 /**
```

```
*

*/

public void getSensors()

{


  try

  {

    //Read the data off of the sensor socket

    sensorData = sensorSockIn.readLine();


    //Print out the information to the screen

    BenderGUI.logReport((“Sensor Data : “ + sensorData ));


  }

  catch(IOException ie)

  {}//end try/catch block


  //Check for valid sensor information

  if(sensorData.length() > 1)
```

```
{
 //Declare an index to reference the distance array
 int index = 0;

 //Tokenize the string data received from the socket
 StringTokenizer st = new StringTokenizer(sensorData, DELIMITTER);

 //Continue this until we've cycled through all of the tokens
 while(st.hasMoreTokens())
 {
  //Get the next token
  String s = st.nextToken();

  //Parse the string to a float and assign the next array
  //element to the float value
  distance[index] = Float.parseFloat(s);

  //Diagnostic print to the user
  BenderGUI.logReport((“Sensor Data = “ + distance[index]));
```

```
      //Move to the next index value
      index++;

    }//end while loop

  }//end if

}//end method getSensors


/**
*
*/
public void close()
{
  try
  {
    sensorSockIn.close();
```

```
            socket.close();

          }

          catch (IOException ie)

          {}


       }//end method close


}//end class SensorInterface
```

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF REFERENCES

Arkin, R. (1999). *Behavior Based Robotics.*

Arkin, R. (1989), "A Motor Schema-Based Mobile Robot Navigation," International Journal of Robotics Research, vol. 8, no. 4, August 1989, pp. 92-112.

Arkin, R. (1990), "Integrating Behavioral, perceptual, and World Knowledge in Reactive Navigation," in Designing Autonomous Agents, P. Maes, ed., Cambridge, Mass., 1990, pp. 105-122.

Balch, T. (2002). *Robot Teams: Polymorphism.*

Beni, G., and Wang, J. (1991), "Theoretical Problems for the Realization of Distributed Robotic System," Proc. 1991 IEEE International Conference on Robotics and Automation, Sacramento, CA, April 1991, pp. 1914-1920.

Brooks, R. (1986), "A Robust Layered control System For a Mobile Robot," IEEE Journal of Robotics and Automation, vol. RA-2, no. 1, March 1986, pp. 14-23.

Brooks, R. (1990), "Elephants Don't Play Chess," in Designing Autonomous Agents, P. Maes, ed., Cambridge, Mass., 1990, pp. 3-15.

Deneubourg J., and others (1990) "Self-Organization Mechanisms in Ant Societies (II): Learning in Foraging and Division of Labor," in From Individual to Collective Behavior in Social Insects, J. M. Pasteels, and J. L. Deneubourg, eds., Experimentia Supplementum, vol. 54, Birkhauser Verlag, Basel, pp. 177-196.

Deneubourg, J., and Goss, S. (1989) "Collective Patterns and Decision Making," Ethology, Ecology and Evolution I, 1989, pp. 295-311.

Feng, D., and Krogh, B. (1989) "A Robust Satisfying Feedback Strategy for Autonomous Navigation," Proc. IEEE International Symposium on Intelligent Control, Albany, NY, September 1989, pp. 379-384.

Genovese, V., and others (1992), "Self-Organizing Behavior and Swarm Intelligence in a Pack of mobile Miniature Robots in Search of Pollutants," Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems, Raleigh, NC, July 1992, pp. 1575-1582.

Habib, M., and others (1992), "Simulation Environment for An Autonomous Decentralized Multi-Agent Robotic system," Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems, Raleigh, NC, July 1992, pp. 1550-1557.

Kaufmann, S. (1991), "Antichaos and Adaptation," Scientific American, August 1991, pp. 78-84.

Kugler, P., and Turvey, M. (1987), Information, Natural Law, and the Self-Assembly of Rhythmic Movement, Hillsdale, NJ: L. Erlbaum Assoc., 1987.

Laird, R. (2000). *Introduction to AI Robotics.*

Miller, D. (1990), "Multiple Behavior-Controlled Micro-Robots for planetary Surface Missions," Proc. 1990 IEEE International Conference on Systems, Man, and Cybernetics, Los Angeles, Ca, November 1990, pp. 281-292.

Nicolis, G., and Progogine, I. (1977), Self-Organization in Nonequilibrium Systems, New York, John Wiley & Sons, 1977.

Partridge, B. (1982), "The structure and Function of Fish Schools," Scientific American, June 1982, pp. 114-123.

Reynolds, C. (1987), "Flocks, Herdes and Schools: A Distributed Behavioral Model," Computer Graphics, vol. 21, no. 4, July 1987, pp. 25-34.

Rodin, E., and Amin, S. (1998), "Intelligent Navigation For an Autonomous Mobile Robot," Proc. International Symposium on Intelligent Control, Arlington, VA, 1998, pp. 366-369.

Selfridge, O. (1962), "The organization of organization," in Self-Organizing Systems, M.C. Yovits, G.T.Jacobi, G.D. Goldstein, eds., Washington D.C., McGregor & Werner, 1962, pp. 1-7.

Stone, P. (2000). *Layered Learning in Multiagent Systems: A Winning Approach to Robotic Soccer.* MIT Press, 2000.

Sugihara, K., and Suzuki, I. (1990), "Distributed Motion Coordination of Multiple Mobile Robot," IEEE International Symposium on Intelligent Control, Philadelphia, PA, September 1990, pp. 138-143.

Theraulaz, G., and others (1990), "Task Differentiation in Polistes Waps Colonies: a Model for Self-Organizing Group of Robots," in From Animals to Animats: Proc. First Int. Conference on Simulation of Adaptive Behavior, J-A. Meyer, and S.W. Wilson, eds., Paris, France, 1990, pp. 346-355.

Wehner, R., and others (1983), "Foraging Strategies in Individually Searching Ants Cataglyphis Bicolor," G. Fisher-Verlag, Stuttgart, 1983.

Woolridge, M. (2001). *Introduction to Multiagent Systems.*

Yovits, M., and others (1962), "Self-Organizing Systems," Self-Organizing Systems, Washington D.C., McGregor & Werner, 1962.

Yuta, S., and Premvuti, S. (1992), "Coordinating Autonomous and Centralized Decision Making to Achieve Cooperative Behaviors Between Multiple Mobile Robots," Proc. 1992 IEEE/RSJ International Conference on Intelligent Robots and Systems, Raleigh, NC, July 1992, pp. 1566-1574.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Bart Everett
   SPAWAR, Systems Center San Diego
   San Diego, California

4. Richard Harkins
   Naval Postgraduate School
   Monterey, California

5. John Hiles
   Naval Postgraduate School
   Monterey, California

6. Katherine Mullens
   SPAWAR, Systems Center San Diego
   San Diego, California